odl Documentation

Release beta

Jonas Adler, Holger Kohr, Ozan Öktem

December 15, 2015

Contents

1 Working with ODL **3**
1.1 As a user . . . 3
1.2 As a developer . . . 3

2 Contents **5**
2.1 User's guide to ODL . . . 5
2.2 Contributing to odl . . . 13
2.3 Frequently asked questions . . . 31
2.4 odl . . . 32

3 Indices and tables **387**

Bibliography **389**

Operator Discretization Library (ODL) is a python library for fast prototyping focusing on (but not restricted to) inverse problems. ODL is being developed at KTH, Royal Institute of Technology.

The main intent of ODL is to enable mathematicians and applied scientists to use different numerical methods on real-world problems without having to implement all necessary parts from the bottom up. ODL provides some of the most heavily used building blocks for numerical algorithms out of the box, which enables users to focus on real scientific issues.

CHAPTER 1

Working with ODL

1.1 As a user

User's guide to ODL

1.2 As a developer

Contributing to odl

CHAPTER 2

Contents

2.1 User's guide to ODL

Welcome to the ODL users guide, this guide is intended to give you a simple introduction to ODL and how to work with it. If you need help on a specific function you should look at its documentation.

2.1.1 Introduction

About ODL

ODL is a Python library for inverse problems. It contains all the abstract mathematical tools needed from analysis, such as sets, vector spaces and operators as well as optimized implementations of many of the most common of these.

Set

A *Set* is the fundamental building block of odl objects. It is intended to mirror the mathematical concept of a set, and has methods to check if an element is contained in the set

```
>>> interv = Interval(0, 1)
>>> 0.5 in interv
True
>>> 2.0 in interv
False
```

The most commonly used sets in odl is the *RealNumbers* the set of all real numbers and *IntervalProd*, arbitrary n-dimensional hypercubes. Several convenience sub-classes such as *Interval*, *Rectangle* and *Cuboid* are also provided.

LinearSpace

A *LinearSpace* is a very important subclass of *Set* and is a general implementation of a mathematical vector space. In odl there are a few kinds of spaces that you will face.

- Continuous function spaces such as *FunctionSpace*, intended to represent mathematical Function space such as the lebesgue space Lp or the space of Linear operators. These are mostly used to represent abstract concepts, and are seldomly used in actual computation.
- $\mathbb{F}^n$ type spaces such as *Rn* and *Cn*, but also the CUDA accelerated version *CudaRn*.

- Discretizations of continous spaces. This may for example be a discretization of a cube using voxels. All discretizations inherit form *Discretization*, but the most important is *DiscreteLp*.
- In addition to this, there are utility spaces such as *ProductSpace* which allows the composition of several spaces into a larger space.

In addition to the spaces, all elements in the spaces inherit from *LinearSpaceVector*. Using these vectors, most standard mathematical operations can be expressed

```
>>> r3 = Rn(3)
>>> x = r3.element([1, 2, 3])
>>> y = r3.element([4, 5, 6])
```

Arithmetic such as addition and multiplication by scalars

```
>>> x + y
Rn(3).element([5.0, 7.0, 9.0])
```

Inner product etc are defined

```
>>> r3.inner(x, y)
32.0
```

See also in depth guide on *LinearSpace*.

Operator

A operator is a function from some *Set* to another. In odl these inherit from the abstract class *Operator*.

See also in depth guide on *Operators*.

Discretizations

Discretizations ...

Diagnostics

Odl also offers tools to verify the correctness of operators and spaces. Examples include verifying that a *Operator*'s derivative is correct:

```
>>> op = MyOperator()
>>> odl.diagnostics.OperatorTest(op).derivative()
```

or verifying that a *LinearSpace* satisfies all expected properties

```
>>> r3 = odl.Rn(5)
>>> odl.diagnostics.SpaceTest(r3).run_tests()
```

See *SpaceTest* and *OperatorTest* for more details.

Installing

Installing odl is intended to be straightforward, and this guide is meant for new users. To install odl you need to do the following steps:

1. *Install Python*

2. *Install Git*
3. *Install ODL*
4. *Run tests*

If you have done any step before, you can ofc skip it.

Install Python

To begin with, you need a python distribution. If you are an experienced user, you can use any distribution you'd like. If you are a python novice, we recommend that you install a full package such as Anaconda.

Anaconda To install Anaconda

1. Download Anaconda from anaconda's webpage
2. Once installed run in a console

```
user$ conda update --all
```

to make sure you have the latest versions of all packages

Install Git

You also need to install Git to be able to download odl.

Overview

Debian / Ubuntu	`sudo apt-get install git`
Fedora	`sudo yum install git`
Windows	Download and install msysGit
OS X	Use the git-osx-installer

In detail See the git page for the most recent information.

Have a look at the github install help pages available from github help

There are good instructions here: http://book.git-scm.com/2_installing_git.html

Install ODL

You are now ready to install ODL! To do that, run the following where you want to install it

```
git clone https://github.com/odlgroup/odl
cd odl
```

For installation in a local user folder run

```
user$ pip install --user .
```

For system-wide installation, run (as root, e.g. using `sudo` or equivalent)

```
root# pip install .
```

(Optional) Install ODLpp

If you also wish to use the (optional) CUDA extensions you need to run

```
user$ git submodule update --init --recursive
user$ cd odlpp
```

From here follow the instructions in odlpp and install it. You then need to re-install ODL.

Run tests

To verify your installation you should run some basic tests. To run these:

```
user$ py.test
```

This requires the module pytest

Getting started

Welcome to the ODL users guide, this guide is intended to give you a simple introduction to ODL and how to work with it. If you need help on a specific function you should look at its documentation.

2.1.2 In depth

This is a more in depth guide to the different parts of odl.

Operators

Operators in ODL are represented by the abstract *Operator* class. As an *abstract class*, it cannot be used directly but must be subclassed for concrete implementation. To define your own operator, you start by writing:

```
class MyOperator(odl.Operator):
    ...
```

Operator has a couple of *abstract methods* which need to be explicitly overridden by any subclass, namely

domain: *Set* Set of elements to which the operator can be applied

range [*Set*] Set in which the operator takes values

As a simple example, you can implement the matrix multiplication operator

$$\mathcal{A} : \mathbb{R}^m \to \mathbb{R}^n, \quad \mathcal{A}(x) = Ax$$

for a matrix $A \in \mathbb{R}^{n \times m}$ as follows:

```
from builtins import super
import numpy as np

class MatVecOperator(odl.Operator):
    def __init__(self, matrix):
        assert isinstance(matrix, np.ndarray)
        self.matrix = matrix
        dom = odl.Rn(matrix.shape[1])
        ran = odl.Rn(matrix.shape[0])
        super().__init__(dom, ran)
```

In addition, an *Operator* needs at least one way of evaluation, *in-place* or *out-of-place*.

In place evaluation

In-place evaluation means that the operator is evaluated on a *Operator.domain* element, and the result is written to an *already existing* *Operator.range* element. To implement this behavior, create the (private) `Operator._apply` method with the following signature, here given for the above example:

```
class MatVecOperator(odl.Operator):
    ...
    def _apply(self, x, out):
        self.matrix.dot(x, out=out.asarray())
```

In-place evaluation is usually more efficient and should be used *whenever possible*.

Out-of-place evaluation

Out-of-place evaluation means that the operator is evaluated on a `domain` element, and the result is written to a *newly allocated* `range` element. To implement this behavior, create the (private) `Operator._call` method with the following signature, here given for the above example:

```
class MatVecOperator(odl.Operator):
    ...
    def _call(self, x):
        return self.range.element(self.matrix.dot(x))
```

Out-of-place evaluation is usually less efficient since it requires allocation of an array and a full copy and should be *generally avoided*.

Important: Do not call these methods directly. Use the call pattern `operator(x)` or `operator(x, out=y)`, e.g.:

```
matrix = np.array([[1, 0], [0, 1], [1, 1]])
operator = MatVecOperator(matrix)
x = odl.Rn(2).one()
y = odl.Rn(3).element()

# Out-of-place evaluation
y = operator(x)

# In-place evaluation
operator(x, out=y)
```

This public calling interface is type-checked, so the private methods do not need to implement type checks.

Operator arithmetic

It is common in applications to perform arithmetic with operators, for example the addition of matrices

$$[A + B]x = Ax + Bx$$

or multiplication of a functional by a scalar

$$[\alpha x^*](x) = \alpha x^*(x)$$

Another example is matrix multiplication, which corresponds to operator composition

$$[AB](x) = A(Bx)$$

All available operator arithmetic is shown below. `A`, `B` represent arbitrary *Operator*'s, `f` is a *Operator* with *Operator.range* a *Field* (sometimes called a functional) and `a` is a scalar.

Code	Meaning	Class
`(A + B)(x)`	`A(x) + B(x)`	*OperatorSum*
`(A * B)(x)`	`A(B(x))`	*OperatorComp*
`(a * A)(x)`	`a * A(x)`	*OperatorLeftScalarMult*
`(A * a)(x)`	`A(a * x)`	*OperatorRightScalarMult*
`(v * f)(x)`	`v * f(x)`	*FunctionalLeftVectorMult*
`(v * A)(x)`	`v * A(x)`	*OperatorLeftVectorMult*
`(A * v)(x)`	`A(v * x)`	*OperatorRightVectorMult*
not available	`A(x) * B(x)`	*OperatorPointwiseProduct*

There are also a few derived expressions using the above:

Code	Meaning
`(+A)(x)`	`A(x)`
`(-A)(x)`	`(-1) * A(x)`
`(A - B)(x)`	`A(x) + (-1) * B(x)`
`A**n(x)`	`A(A**(n-1)(x)), A^1(x) = A(x)`
`(A / a)(x)`	`A((1/a) * x)`
`(A @ B)(x)`	`(A * B)(x)`

Except for composition, operator arithmetic is generally only defined when *Operator.domain* and *Operator.range* is a *LinearSpace* or *Field*.

LinearSpace

The *LinearSpace* represent abstract mathematical concepts of vector spaces. It cannot be used directly but are rather intended to be subclassed by concrete space implementations. The space provide default implementations of the most important vector space operations. See the documentation of the respective classes for more details.

The concept of linear vector spaces in ODL is largely inspired by the Rice Vector Library (RVL).

The abstract *LinearSpace* class is intended for quick prototyping. It has a number of abstract methods which must be overridden by a subclass. On the other hand, it provides automatic error checking and numerous attributes and methods for convenience.

Abstract methods

In the following, the abstract methods are explained in detail.

Element creation `element(inp=None)`

This public method is the factory for the inner *LinearSpaceVector* class. It creates a new element of the space, either from scratch or from an existing data container. In the simplest possible case, it just delegates the construction to the *LinearSpaceVector* class.

If no data is provided, the new element is **merely allocated, not initialized**, thus it can contain *any* value.

Parameters:

inp `[object, optional]` A container for values for the element initialization

Returns:

element [*LinearSpaceVector*] The new vector

Linear combination `_lincomb(a, x1, b, x2, out)`

This private method is the raw implementation (i.e. without error checking) of the linear combination `out = a * x1 + b * x2`. *LinearSpace._lincomb* and its public counterpart *LinearSpace.lincomb* are used to covera range of convenience functions, see below.

Parameters:

a, b [scalars, must be members of the space's `field`] Multiplicative scalar factors for input vector `x1` or `x2`, respectively

x1, x2 [*LinearSpaceVector*] Input vectors

out [*LinearSpaceVector*] Element to which the result of the computation is written

Returns: `None`

Requirements:

- Aliasing of `x1`, `x2` and `out` **must** be allowed.
- The input vectors `x1` and `x2` **must not** be modified.
- The initial state of the output vector `out` **must not** influence the result.

Underlying scalar field `field`

The public attribute determining the type of scalars which underlie the space. Can be instances of either *RealNumbers* or *ComplexNumbers* (see *Field*).

Should be implemented as a `@property` to make it immutable.

Equality check `__eq__(other)`

LinearSpace inherits this abstract method from *Set*. Its purpose is to check two *LinearSpace* instances for equality.

Parameters:

other [`object`] The object to compare to

Returns:

equals [`bool`] `True` if `other` is the same *LinearSpace*, `False` otherwise

Distance (optional) `_dist(x1, x2)`

A raw (not type-checking) private method measuring the distance between two vectors `x1` and `x2`.

A space with a distance is called a **metric space**.

Parameters:

x1,x2 [*LinearSpaceVector*] Vectors whose mutual distance to calculate

Returns:

distance [`float`] The distance between `x1` and `x2`, measured in the space's metric

Requirements:

- `_dist(x, y) == _dist(y, x)`
- `_dist(x, y) <= _dist(x, z) + _dist(z, y)`
- `_dist(x, y) >= 0`
- `_dist(x, y) == 0` (approx.) if and only if `x == y` (approx.)

Norm (optional) `_norm(x)`

A raw (not type-checking) private method measuring the length of a space element `x`.

A space with a norm is called a **normed space**.

Parameters:

x [*LinearSpaceVector*] The vector to measure

Returns:

norm [`float`] The length of `x` as measured in the space's norm

Requirements:

- `_norm(s * x) = |s| * _norm(x)` for any scalar `s`
- `_norm(x + y) <= _norm(x) + _norm(y)`
- `_norm(x) >= 0`
- `_norm(x) == 0` (approx.) if and only if `x == 0` (approx.)

Inner product (optional) `_inner(x, y)`

A raw (not type-checking) private method calculating the inner product of two space elements `x` and `y`.

Parameters:

x,y [*LinearSpaceVector*] Vectors whose inner product to calculate

Returns:

inner [`float` or `complex`] The inner product of `x` and `y`. If *LinearSpace.field* is the set of real numbers, `inner` is a `float`, otherwise `complex`.

Requirements:

- `_inner(x, y) == _inner(y, x)^*` with '*' = complex conjugation
- `_inner(s * x, y) == s * _inner(x, y)` for `s` scalar
- `_inner(x + z, y) == _inner(x, y) + _inner(z, y)`
- `_inner(x, x) == 0` (approx.) if and only if `x == 0` (approx.)

Pointwise multiplication (optional) `_multiply(x1, x2, out)`

A raw (not type-checking) private method multiplying two vectors `x1` and `x2` element-wise and storing the result in `out`.

Parameters:

x1, x2 [*LinearSpaceVector*] Vectors whose element-wise product to calculate

out [*LinearSpaceVector*] Vector to store the result

Returns: `None`

Requirements:

- `_multiply(x, y, out) <==> _multiply(y, x, out)`
- **`_multiply(s * x, y, out) <==> _multiply(x, y, out); out *= s <==>`** `_multiply(x, s * y, out)` for any scalar `s`
- There is a space element `one` with `out` after `_multiply(one, x, out)` or `_multiply(x, one, out)` equals `x`.

Notes

- A normed space is automatically a metric space with the distance function `_dist(x, y) = _norm(x - y)`.
- A Hilbert space (inner product space) is automatically a normed space with the norm function `_norm(x) = sqrt(_inner(x, x))`.
- The conditions on the pointwise multiplication constitute a *unital commutative algebra* in the mathematical sense.

References

See Wikipedia's mathematical overview articles Vector space, Algebra.

Discretizations

TODO.

References

See Wikipedia's mathematical overview articles Discretization.

2.2 Contributing to odl

2.2.1 How to document

ODL is documented using sphinx and a modified version of numpydoc. An example documentation is given below

```
class MyClass(object):

    """Calculate important things.

    The first line summarizes the class, after that comes a blank
    line followed by a more detailed description (both optional).
    Confine the docstring to 72 characters per line. In general, try
    to follow `PEP257`_ in the docstring style.

    Docstrings can have sections with headers, signalized by a
    single-dash underline, e.g. "References". Check out
    `Numpydoc`_ for the recognized section labels.
```

```
    References
    ----------
    .. _PEP257: https://www.python.org/dev/peps/pep-0257/
    .. _Numpydoc: https://github.com/numpy/numpy/blob/master/doc/\
       HOWTO_DOCUMENT.rst.txt
    """

    def __init__(self, c, parameter=None):
        """Initializer doc goes here.

        Parameters
        ----------
        c : `float`
            Constant to scale by
        parameter : `float`, optional
            Some extra parameter
        """
        self.c = c
        self.parameter = parameter

    def my_method(self, x, y):
        """Calculate ``c * (x + y).``

        The first row is a summary, after that goes
        a more detailed description.

        Parameters
        ----------
        x : `float`
            First summand
        y : `float`
            Second summand

        Returns
        -------
        scaled_sum : `float`
            Result of ``c * (x + y)``

        Examples
        --------
        Examples should be working pieces of code and are checked with
        ``doctest`` for consistent output.

        >>> obj = MyClass(5)
        >>> obj(3, 5)
        8.0
        """
        return self.c * (x + y)
```

Some short tips

- Text within backticks: `` `some_target` `` will create a link to the target.
- Make sure that the first line is short and descriptive.
- Examples are often better than long descriptions.

Quick summary of PEP257

- Write docstrings always with triple double quotes `"""`, even one-liners

- Class docstrings are separated from the class definition line by a blank line, functions and methods begin directly in the next line.
- Use imperative style ("Calculate", not "Calculates") in the summary (=first) line and end it with a full stop. Do not add a space after the opening triple quotes.
- For one-liners: put the closing quotes on the same line. Otherwise: make a new line for the closing quotes.
- Document at least all *public* methods and attributes.

Advanced

This section covers advanced topics for developers that need to change how the doc works.

Re-generating the doc

Autosummary currently does not support nested modules, so to handle this we auto-generate `.rst` files for each module. This is done using the `doc/source/generate_doc.py` script.

Modifications to numpydoc

Numpydoc has been modified in the following ways:

- The numpy sphinx domain has been removed.
- More `extra_public_methods` added.
- `:autoclass:` summaries now link to full name, which allows subclassing between packages.

2.2.2 Working with *odl* source code

Contents:

Introduction

These pages describe a git and github workflow for the odl project.

There are several different workflows here, for different ways of working with *odl*.

This is not a comprehensive git reference, it's just a workflow for our own project. It's tailored to the github hosting service. You may well find better or quicker ways of getting stuff done with git, but these should get you started.

For general resources for learning git, see *git resources*.

Install git

Overview

Debian / Ubuntu	`sudo apt-get install git`
Fedora	`sudo yum install git`
Windows	Download and install msysGit
OS X	Use the git-osx-installer

In detail

See the git page for the most recent information.

Have a look at the github install help pages available from github help

There are good instructions here: http://book.git-scm.com/2_installing_git.html

Following the latest source

These are the instructions if you just want to follow the latest *odl* source, but you don't need to do any development for now.

The steps are:

- *Install git*
- get local copy of the odl github git repository
- update local copy from time to time

Get the local copy of the code

From the command line:

```
git clone git://github.com/odlgroup/odl.git
```

You now have a copy of the code tree in the new `odl` directory.

Updating the code

From time to time you may want to pull down the latest code. Do this with:

```
cd odl
git pull
```

The tree in `odl` will now have the latest changes from the initial repository.

Making a patch

You've discovered a bug or something else you want to change in odl .. — excellent!

You've worked out a way to fix it — even better!

You want to tell us about it — best of all!

The easiest way is to make a *patch* or set of patches. Here we explain how. Making a patch is the simplest and quickest, but if you're going to be doing anything more than simple quick things, please consider following the *Git for development* model instead.

Making patches

Overview

```
# tell git who you are
git config --global user.email you@yourdomain.example.com
git config --global user.name "Your Name Comes Here"
# get the repository if you don't have it
git clone git://github.com/odlgroup/odl.git
# make a branch for your patching
cd odl
git branch the-fix-im-thinking-of
git checkout the-fix-im-thinking-of
# hack, hack, hack
# Tell git about any new files you've made
git add somewhere/tests/test_my_bug.py
# commit work in progress as you go
git commit -am 'BF - added tests for Funny bug'
# hack hack, hack
git commit -am 'BF - added fix for Funny bug'
# make the patch files
git format-patch -M -C master
```

Then, send the generated patch files to the odl mailing list — where we will thank you warmly.

In detail

1. Tell git who you are so it can label the commits you've made:

```
git config --global user.email you@yourdomain.example.com
git config --global user.name "Your Name Comes Here"
```

2. If you don't already have one, clone a copy of the odl repository:

```
git clone git://github.com/odlgroup/odl.git
cd odl
```

3. Make a 'feature branch'. This will be where you work on your bug fix. It's nice and safe and leaves you with access to an unmodified copy of the code in the main branch:

```
git branch the-fix-im-thinking-of
git checkout the-fix-im-thinking-of
```

4. Do some edits, and commit them as you go:

```
# hack, hack, hack
# Tell git about any new files you've made
git add somewhere/tests/test_my_bug.py
# commit work in progress as you go
git commit -am 'BF - added tests for Funny bug'
# hack hack, hack
git commit -am 'BF - added fix for Funny bug'
```

   Note the `-am` options to `commit`. The `m` flag just signals that you're going to type a message on the command line. The `a` flag — you can just take on faith — or see why the -a flag?.

5. When you have finished, check you have committed all your changes:

```
git status
```

6. Finally, make your commits into patches. You want all the commits since you branched from the `master` branch:

```
    git format-patch -M -C master
```

You will now have several files named for the commits:

```
    0001-BF-added-tests-for-Funny-bug.patch
    0002-BF-added-fix-for-Funny-bug.patch
```

Send these files to the odl mailing list.

When you are done, to switch back to the main copy of the code, just return to the `master` branch:

```
git checkout master
```

Moving from patching to development

If you find you have done some patches, and you have one or more feature branches, you will probably want to switch to development mode. You can do this with the repository you have.

Fork the odl repository on github — *Making your own copy (fork) of odl*. Then:

```
# checkout and refresh master branch from main repo
git checkout master
git pull origin master
# rename pointer to main repository to 'upstream'
git remote rename origin upstream
# point your repo to default read / write to your fork on github
git remote add origin git@github.com:your-user-name/odl.git
# push up any branches you've made and want to keep
git push origin the-fix-im-thinking-of
```

Then you can, if you want, follow the *Development workflow*.

Git for development

Contents:

Making your own copy (fork) of odl

You need to do this only once. The instructions here are very similar to the instructions at http://help.github.com/forking/ — please see that page for more detail. We're repeating some of it here just to give the specifics for the odl project, and to suggest some default names.

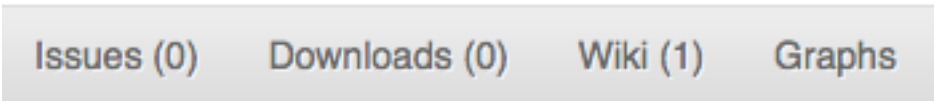
Set up and configure a github account If you don't have a github account, go to the github page, and make one.

You then need to configure your account to allow write access — see the `Generating SSH keys` help on github help.

Create your own forked copy of odl

1. Log into your github account.
2. Go to the odl github home at odl github.

3. Click on the *fork* button:



Now, after a short pause and some 'Hardcore forking action', you should find yourself at the home page for your own forked copy of odl.

Set up your fork

First you follow the instructions for *Making your own copy (fork) of odl*.

Overview

```
git clone git@github.com:your-user-name/odl.git
cd odl
git remote add upstream git://github.com/odlgroup/odl.git
```

In detail

Clone your fork

1. Clone your fork to the local computer with `git clone git@github.com:your-user-name/odl.git`
2. Investigate. Change directory to your new repo: `cd odl`. Then `git branch -a` to show you all branches. You'll get something like:

```
* master
remotes/origin/master
```

   This tells you that you are currently on the `master` branch, and that you also have a `remote` connection to `origin/master`. What remote repository is `remote/origin`? Try `git remote -v` to see the URLs for the remote. They will point to your github fork.

   Now you want to connect to the upstream odl github repository, so you can merge in changes from trunk.

Linking your repository to the upstream repo

```
cd odl
git remote add upstream git://github.com/odlgroup/odl.git
```

`upstream` here is just the arbitrary name we're using to refer to the main odl repository at odl github.

Note that we've used `git://` for the URL rather than `git@`. The `git://` URL is read only. This means we that we can't accidentally (or deliberately) write to the upstream repo, and we are only going to use it to merge into our own code.

Just for your own satisfaction, show yourself that you now have a new 'remote', with `git remote -v show`, giving you something like:

```
upstream     git://github.com/odlgroup/odl.git (fetch)
upstream     git://github.com/odlgroup/odl.git (push)
origin       git@github.com:your-user-name/odl.git (fetch)
origin       git@github.com:your-user-name/odl.git (push)
```

Configure git

Overview Your personal git configurations are saved in the `.gitconfig` file in your home directory.

Here is an example `.gitconfig` file:

```
[user]
        name = Your Name
        email = you@yourdomain.example.com

[alias]
        ci = commit -a
        co = checkout
        st = status
        stat = status
        br = branch
        wdiff = diff --color-words

[core]
        editor = vim

[merge]
        summary = true
```

You can edit this file directly or you can use the `git config --global` command:

```
git config --global user.name "Your Name"
git config --global user.email you@yourdomain.example.com
git config --global alias.ci "commit -a"
git config --global alias.co checkout
git config --global alias.st "status -a"
git config --global alias.stat "status -a"
git config --global alias.br branch
git config --global alias.wdiff "diff --color-words"
git config --global core.editor vim
git config --global merge.summary true
```

To set up on another computer, you can copy your `~/.gitconfig` file, or run the commands above.

In detail

user.name and user.email It is good practice to tell git who you are, for labeling any changes you make to the code. The simplest way to do this is from the command line:

```
git config --global user.name "Your Name"
git config --global user.email you@yourdomain.example.com
```

This will write the settings into your git configuration file, which should now contain a user section with your name and email:

```
[user]
      name = Your Name
      email = you@yourdomain.example.com
```

Of course you'll need to replace `Your Name` and `you@yourdomain.example.com` with your actual name and email address.

Aliases You might well benefit from some aliases to common commands.

For example, you might well want to be able to shorten `git checkout` to `git co`. Or you may want to alias `git diff --color-words` (which gives a nicely formatted output of the diff) to `git wdiff`

The following `git config --global` commands:

```
git config --global alias.ci "commit -a"
git config --global alias.co checkout
git config --global alias.st "status -a"
git config --global alias.stat "status -a"
git config --global alias.br branch
git config --global alias.wdiff "diff --color-words"
```

will create an `alias` section in your `.gitconfig` file with contents like this:

```
[alias]
        ci = commit -a
        co = checkout
        st = status -a
        stat = status -a
        br = branch
        wdiff = diff --color-words
```

Editor You may also want to make sure that your editor of choice is used

```
git config --global core.editor vim
```

Merging To enforce summaries when doing merges (`~/.gitconfig` file again):

```
[merge]
   log = true
```

Or from the command line:

```
git config --global merge.log true
```

Fancy log output This is a very nice alias to get a fancy log output; it should go in the `alias` section of your `.gitconfig` file:

```
lg = log --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr) %C(bold blue)[%
```

You use the alias with:

```
git lg
```

and it gives graph / text output something like this (but with color!):

```
* 6d8e1ee - (HEAD, origin/my-fancy-feature, my-fancy-feature) NF - a fancy file (45 minutes ago) [Mat
*   d304a73 - (origin/placeholder, placeholder) Merge pull request #48 from hhuuggoo/master (2 weeks
|\
| * 4aff2a8 - fixed bug 35, and added a test in test_bugfixes (2 weeks ago) [Hugo]
|/
* a7ff2e5 - Added notes on discussion/proposal made during Data Array Summit. (2 weeks ago) [Corran W
* 68f6752 - Initial implimentation of AxisIndexer - uses 'index_by' which needs to be changed to a ca
*   376adbd - Merge pull request #46 from terhorst/master (2 weeks ago) [Jonathan Terhorst]
|\
| * b605216 - updated joshu example to current api (3 weeks ago) [Jonathan Terhorst]
| * 2e991e8 - add testing for outer ufunc (3 weeks ago) [Jonathan Terhorst]
| * 7beda5a - prevent axis from throwing an exception if testing equality with non-axis object (3 wee
| * 65af65e - convert unit testing code to assertions (3 weeks ago) [Jonathan Terhorst]
| *   956fbab - Merge remote-tracking branch 'upstream/master' (3 weeks ago) [Jonathan Terhorst]
| |\
| |/
```

Thanks to Yury V. Zaytsev for posting it.

Development workflow

You already have your own forked copy of the odl repository, by following *Making your own copy (fork) of odl*. You have *Set up your fork*. You have configured git by following *Configure git*. Now you are ready for some real work.

Workflow summary In what follows we'll refer to the upstream odl `master` branch, as "trunk".

- Don't use your `master` branch for anything. Consider deleting it.
- When you are starting a new set of changes, fetch any changes from trunk, and start a new *feature branch* from that.
- Make a new branch for each separable set of changes — "one task, one branch" (ipython git workflow).
- Name your branch for the purpose of the changes - e.g. `bugfix-for-issue-14` or `refactor-database-code`.
- If you can possibly avoid it, avoid merging trunk or any other branches into your feature branch while you are working.
- If you do find yourself merging from trunk, consider *Rebasing on trunk*
- Ask on the odl mailing list if you get stuck.
- Ask for code review!

This way of working helps to keep work well organized, with readable history. This in turn makes it easier for project maintainers (that might be you) to see what you've done, and why you did it.

See linux git workflow and ipython git workflow for some explanation.

Consider deleting your master branch It may sound strange, but deleting your own `master` branch can help reduce confusion about which branch you are on. See deleting master on github for details.

Update the mirror of trunk First make sure you have done *Linking your repository to the upstream repo*.

From time to time you should fetch the upstream (trunk) changes from github:

```
git fetch upstream
```

This will pull down any commits you don't have, and set the remote branches to point to the right commit. For example, 'trunk' is the branch referred to by (remote/branchname) `upstream/master` - and if there have been commits since you last checked, `upstream/master` will change after you do the fetch.

Make a new feature branch When you are ready to make some changes to the code, you should start a new branch. Branches that are for a collection of related edits are often called 'feature branches'.

Making an new branch for each set of related changes will make it easier for someone reviewing your branch to see what you are doing.

Choose an informative name for the branch to remind yourself and the rest of us what the changes in the branch are for. For example `add-ability-to-fly`, or `buxfix-for-issue-42`.

```
# Update the mirror of trunk
git fetch upstream
# Make new feature branch starting at current trunk
git branch my-new-feature upstream/master
git checkout my-new-feature
```

Generally, you will want to keep your feature branches on your public github fork of odl. To do this, you git push this new branch up to your github repo. Generally (if you followed the instructions in these pages, and by default), git will have a link to your github repo, called `origin`. You push up to your own repo on github with:

```
git push origin my-new-feature
```

In git >= 1.7 you can ensure that the link is correctly set by using the `--set-upstream` option:

```
git push --set-upstream origin my-new-feature
```

From now on git will know that `my-new-feature` is related to the `my-new-feature` branch in the github repo.

The editing workflow

Overview

```
# hack hack
git add my_new_file
git commit -am 'NF - some message'
git push
```

In more detail

1. Make some changes
2. See which files have changed with `git status` (see git status). You'll see a listing like this one:

```
    # On branch ny-new-feature
    # Changed but not updated:
    #   (use "git add <file>..." to update what will be committed)
    #   (use "git checkout -- <file>..." to discard changes in working directory)
    #
    #  modified:   README
    #
    # Untracked files:
    #   (use "git add <file>..." to include in what will be committed)
```

```
    #
    #  INSTALL
    no changes added to commit (use "git add" and/or "git commit -a")
```

3. Check what the actual changes are with `git diff` (git diff).
4. Add any new files to version control `git add new_file_name` (see git add).
5. To commit all modified files into the local copy of your repo, do `git commit -am 'A commit message'`. Note the `-am` options to `commit`. The `m` flag just signals that you're going to type a message on the command line. The `a` flag — you can just take on faith — or see why the -a flag? — and the helpful use-case description in the tangled working copy problem. The git commit manual page might also be useful.
6. To push the changes up to your forked repo on github, do a `git push` (see git push).

The commit message Bear in mind that the commit message will be part of the history of the repository, shown by typing `git log`, so good messages will make the history searchable. Don't see the commit message as an annoyance, but rather as an important part of your contribution.

We appreciate if you follow the following style:

1. Start your commit with an acronym, e.g., `BUG`, `TST` or `STY` to indicate what kind of modification you make.
2. Write a one-line summary of your modification no longer than 50 characters. If you have a hard time summarizing you changes, maybe you need to split up the commit into parts.

   Use imperative style, i.e. write `add super feature` or `fix horrific bug` rather than `added`, `fixed` .... This saves two characters for something else.

   Don't use markdown. You can refer to issues by writing `#12`. You can even close an issue by writing `closes #12`, but do that only if you are sure.
3. (optional) Write an extended summary. Describe why these changes are necessary and what the new code does better than the old one. You can use markdown here, i.e. create lists, tables, ...

Ask for your changes to be reviewed or merged When you are ready to ask for someone to review your code and consider a merge:

1. Go to the URL of your forked repo, say `http://github.com/your-user-name/odl`.
2. Use the 'Switch Branches' dropdown menu near the top left of the page to select the branch with your changes:

3. Click on the 'Pull request' button:

Enter a title for the set of changes, and some explanation of what you've done. Say if there is anything you'd like particular attention for - like a complicated change or some code you are not happy with.

If you don't think your request is ready to be merged, just say so in your pull request message. This is still a good way of getting some preliminary code review.

Some other things you might want to do

Delete a branch on github

```
git checkout master
# delete branch locally
git branch -D my-unwanted-branch
# delete branch on github
git push origin :my-unwanted-branch
```

(Note the colon `:` before `test-branch`. See also: http://github.com/guides/remove-a-remote-branch

Several people sharing a single repository If you want to work on some stuff with other people, where you are all committing into the same repository, or even the same branch, then just share it via github.

First fork odl into your account, as from *Making your own copy (fork) of odl*.

Then, go to your forked repository github page, say `http://github.com/your-user-name/odl`

Click on the 'Admin' button, and add anyone else to the repo as a collaborator:

Now all those people can do:

```
git clone git@githhub.com:your-user-name/odl.git
```

Remember that links starting with `git@` use the ssh protocol and are read-write; links starting with `git://` are read-only.

Your collaborators can then commit directly into that repo with the usual:

```
git commit -am 'ENH - much better code'
git push origin master # pushes directly into your repo
```

Explore your repository To see a graphical representation of the repository branches and commits:

```
gitk --all
```

To see a linear list of commits for this branch:

```
git log
```

You can also look at the network graph visualizer for your github repo.

Finally the *Fancy log output* `lg` alias will give you a reasonable text-based graph of the repository.

Rebasing on trunk Let's say you thought of some work you'd like to do. You *Update the mirror of trunk* and *Make a new feature branch* called `cool-feature`. At this stage trunk is at some commit, let's call it E. Now you make some new commits on your `cool-feature` branch, let's call them A, B, C. Maybe your changes take a while, or you come back to them after a while. In the meantime, trunk has progressed from commit E to commit (say) G:

```
      A---B---C cool-feature
     /
D---E---F---G trunk
```

At this stage you consider merging trunk into your feature branch, and you remember that this here page sternly advises you not to do that, because the history will get messy. Most of the time you can just ask for a review, and not worry that trunk has got a little ahead. But sometimes, the changes in trunk might affect your changes, and you need to harmonize them. In this situation you may prefer to do a rebase.

rebase takes your changes (A, B, C) and replays them as if they had been made to the current state of `trunk`. In other words, in this case, it takes the changes represented by A, B, C and replays them on top of G. After the rebase, your history will look like this:

```
              A'--B'--C' cool-feature
             /
D---E---F---G trunk
```

See rebase without tears for more detail.

To do a rebase on trunk:

```
# Update the mirror of trunk
git fetch upstream
# go to the feature branch
git checkout cool-feature
# make a backup in case you mess up
git branch tmp cool-feature
# rebase cool-feature onto trunk
git rebase --onto upstream/master upstream/master cool-feature
```

In this situation, where you are already on branch `cool-feature`, the last command can be written more succinctly as:

```
git rebase upstream/master
```

When all looks good you can delete your backup branch:

```
git branch -D tmp
```

If it doesn't look good you may need to have a look at *Recovering from mess-ups*.

If you have made changes to files that have also changed in trunk, this may generate merge conflicts that you need to resolve - see the git rebase man page for some instructions at the end of the "Description" section. There is some related help on merging in the git user manual - see resolving a merge.

Recovering from mess-ups Sometimes, you mess up merges or rebases. Luckily, in git it is relatively straightforward to recover from such mistakes.

If you mess up during a rebase:

```
git rebase --abort
```

If you notice you messed up after the rebase:

```
# reset branch back to the saved point
git reset --hard tmp
```

If you forgot to make a backup branch:

```
# look at the reflog of the branch
git reflog show cool-feature

8630830 cool-feature@{0}: commit: BUG: io: close file handles immediately
278dd2a cool-feature@{1}: rebase finished: refs/heads/my-feature-branch onto 11ee694744f2552d
26aa21a cool-feature@{2}: commit: BUG: lib: make seek_gzip_factory not leak gzip obj
...

# reset the branch to where it was before the botched rebase
git reset --hard cool-feature@{2}
```

Rewriting commit history

Note: Do this only for your own feature branches.

There's an embarrassing typo in a commit you made? Or perhaps the you made several false starts you would like the posterity not to see.

This can be done via *interactive rebasing*.

Suppose that the commit history looks like this:

```
git log --oneline
eadc391 Fix some remaining bugs
a815645 Modify it so that it works
2dec1ac Fix a few bugs + disable
13d7934 First implementation
6ad92e5 * masked is now an instance of a new object, MaskedConstant
29001ed Add pre-nep for a copule of structured_array_extensions.
...
```

and `6ad92e5` is the last commit in the `cool-feature` branch. Suppose we want to make the following changes:

- Rewrite the commit message for `13d7934` to something more sensible.
- Combine the commits `2dec1ac`, `a815645`, `eadc391` into a single one.

We do as follows:

```
# make a backup of the current state
git branch tmp HEAD
# interactive rebase
git rebase -i 6ad92e5
```

This will open an editor with the following text in it:

```
pick 13d7934 First implementation
pick 2dec1ac Fix a few bugs + disable
```

```
pick a815645 Modify it so that it works
pick eadc391 Fix some remaining bugs

# Rebase 6ad92e5..eadc391 onto 6ad92e5
#
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
#
# If you remove a line here THAT COMMIT WILL BE LOST.
# However, if you remove everything, the rebase will be aborted.
#
```

To achieve what we want, we will make the following changes to it:

```
r 13d7934 First implementation
pick 2dec1ac Fix a few bugs + disable
f a815645 Modify it so that it works
f eadc391 Fix some remaining bugs
```

This means that (i) we want to edit the commit message for `13d7934`, and (ii) collapse the last three commits into one. Now we save and quit the editor.

Git will then immediately bring up an editor for editing the commit message. After revising it, we get the output:

```
[detached HEAD 721fc64] FOO: First implementation
 2 files changed, 199 insertions(+), 66 deletions(-)
[detached HEAD 0f22701] Fix a few bugs + disable
 1 files changed, 79 insertions(+), 61 deletions(-)
Successfully rebased and updated refs/heads/my-feature-branch.
```

and the history looks now like this:

```
0f22701 Fix a few bugs + disable
721fc64 ENH: Sophisticated feature
6ad92e5 * masked is now an instance of a new object, MaskedConstant
```

If it went wrong, recovery is again possible as explained *above*.

Maintainer workflow

This page is for maintainers — those of us who merge our own or other peoples' changes into the upstream repository.

Being as how you're a maintainer, you are completely on top of the basic stuff in *Development workflow*.

The instructions in *Linking your repository to the upstream repo* add a remote that has read-only access to the upstream repo. Being a maintainer, you've got read-write access.

It's good to have your upstream remote have a scary name, to remind you that it's a read-write remote:

```
git remote add upstream-rw git@github.com:odlgroup/odl.git
git fetch upstream-rw
```

Integrating changes Let's say you have some changes that need to go into trunk (`upstream-rw/master`).

The changes are in some branch that you are currently on. For example, you are looking at someone's changes like this:

```
git remote add someone git://github.com/someone/odl.git
git fetch someone
git branch cool-feature --track someone/cool-feature
git checkout cool-feature
```

So now you are on the branch with the changes to be incorporated upstream. The rest of this section assumes you are on this branch.

A few commits If there are only a few commits, consider rebasing to upstream:

```
# Fetch upstream changes
git fetch upstream-rw
# rebase
git rebase upstream-rw/master
```

Remember that, if you do a rebase, and push that, you'll have to close any github pull requests manually, because github will not be able to detect the changes have already been merged.

A long series of commits If there are a longer series of related commits, consider a merge instead:

```
git fetch upstream-rw
git merge --no-ff upstream-rw/master
```

The merge will be detected by github, and should close any related pull requests automatically.

Note the `--no-ff` above. This forces git to make a merge commit, rather than doing a fast-forward, so that these set of commits branch off trunk then rejoin the main history with a merge, rather than appearing to have been made directly on top of trunk.

Check the history Now, in either case, you should check that the history is sensible and you have the right commits:

```
git log --oneline --graph
git log -p upstream-rw/master..
```

The first line above just shows the history in a compact way, with a text representation of the history graph. The second line shows the log of commits excluding those that can be reached from trunk (`upstream-rw/master`), and including those that can be reached from current HEAD (implied with the `..` at the end). So, it shows the commits unique to this branch compared to trunk. The `-p` option shows the diff for these commits in patch form.

Push to trunk

```
git push upstream-rw my-new-feature:master
```

This pushes the `my-new-feature` branch in this repository to the `master` branch in the `upstream-rw` repository.

git resources

Tutorials and summaries

- github help has an excellent series of how-to guides.

- learn.github has an excellent series of tutorials
- The pro git book is a good in-depth book on git.
- A git cheat sheet is a page giving summaries of common commands.
- The git user manual
- The git tutorial
- The git community book
- git ready — a nice series of tutorials
- git casts — video snippets giving git how-tos.
- git magic — extended introduction with intermediate detail
- The git parable is an easy read explaining the concepts behind git.
- git foundation expands on the git parable.
- Fernando Perez' git page — Fernando's git page — many links and tips
- A good but technical page on git concepts
- git svn crash course: git for those of us used to subversion

Advanced git workflow

There are many ways of working with git; here are some posts on the rules of thumb that other projects have come up with:

- Linus Torvalds on git management
- Linus Torvalds on linux git workflow . Summary; use the git tools to make the history of your edits as clean as possible; merge from upstream edits as little as possible in branches where you are doing active development.

Manual pages online

You can get these on your own machine with (e.g) `git help push` or (same thing) `git push --help`, but, for convenience, here are the online manual pages for some common commands:

- git add
- git branch
- git checkout
- git clone
- git commit
- git config
- git diff
- git log
- git pull
- git push
- git remote
- git status

2.3 Frequently asked questions

Abbreviations: **Q** uestion – **P** roblem – **S** olution

2.3.1 General errors

1. **Q:** When importing `odl`, the following error is shown:

```
File "/path/to/odl/odl/__init__.py", line 36

  from . import diagnostics

  ImportError: cannot import diagnostics
```

P: When you for the first time import (=execute) a module or execute a script, a bytecode file is created, basically to speed up execution next time. If you installed `odl` with `pip -e` (`--editable`), these files can interfere with changes to your codebase.

S: Delete the bytecode files. In a standard GNU/Linux shell, you can simply invoke (in your `odl` working directory)

```
find . -name *.pyc | xargs rm
```

2.3.2 Errors related to Python 2/3

1. **Q:** I follow your recommendation to call `super().__init__(dom, ran)` in the `__init__()` method of `MyOperator`, but I get the following error:

```
File <...>, line ..., in __init__
        super().__init__(dom, ran)

TypeError: super() takes at least 1 argument (0 given)
```

P: The `super()` function in Python 2 has to be called with a type as first argument, whereas in Python 3, the type argument is optional and usually not needed.

S: We recommend to include `from builtins import super` in your module to backport the new Py3 `super()` function. This way, your code will run in both Python 2 and 3.

2.3.3 Usage

1. **Q:** I want to write an *`Operator`* of with two input arguments, for example

$$f(x, y) \to x + y$$

But ODL only supports single arguments. How do I do this?

P: Mathematically, such a operator is defined

$$X \times Y \to Z$$

ODL adhers to the strict definition of this and hence only takes one parameter $x \in X \times Y$. This x is then a tuple of elements $x = [x_1, x_2], x_1 \in X, x_2 \in Y$.

S: The solution is to make the domain of the operator a *ProductSpace*, if X and Y are *LinearSpace*:s, or a *CartesianProduct* if they are simply *Set*:s. Thus you can do

$$f([x, y]) \to x + y$$

2.4 odl

ODL is a functional analysis library with a focus on discretization.

ODL suppors abstract sets, linear vector spaces defined on such and Operators/Functionals defined on these sets. It is intended to be used to write general code and faciliate code reuse.

Modules

2.4.1 diagnostics

Automated tests for ODL.

Modules

examples

Functions for generating standardized examples in spaces.

Functions

samples(*sets)	Generate some samples from the given sets.
scalar_examples(field)	Generate example scalars in `field`.
vector_examples(space)	Generate example vectors in `space`.

samples

odl.diagnostics.examples.**samples**(**sets*)

Generate some samples from the given sets.

Currently supports vectors according to *vector_examples* and scalars according to *scalar_examples*.

Parameters*sets : *Set* instance(s)

Returnssamples : `generator`

Generator that yields tuples of examples from the sets.

scalar_examples

odl.diagnostics.examples.**scalar_examples**(*field*)

Generate example scalars in `field`.

Parametersfield : *Field*

The field to generate examples from

Returnsexamples : `generator`

Yields elements in `field`

vector_examples

`odl.diagnostics.examples.`**`vector_examples`**(*space*)
Generate example vectors in `space`.

Parametersspace : *LinearSpace*

The space to generate examples from

Returnsexamples : `generator`

Yields tuples (`string`, *LinearSpaceVector*) where `string` is a short description of the vector

operator

Standardized tests for *Operator*'s.

Classes

OperatorTest(operator[, operator_norm])	Automated tests for *Operator* implementations.

OperatorTest

class `odl.diagnostics.operator.`**`OperatorTest`**(*operator*, *operator_norm=None*)
Bases: `object`

Automated tests for *Operator* implementations.

This class allows users to automatically test various features of an Operator such as linearity and the adjoint definition.

Methods

`__eq__`	Return self==value.
adjoint()	Verify that *Operator.adjoint* works appropriately.
derivative([step])	Verify that *Operator.derivative* works appropriately.
linear()	Verify that the operator is actually linear.
norm()	Estimate the operator norm of the operator.
run_tests()	Run all tests on this operator.
self_adjoint()	Verify (Ax, y) = (x, Ay)

OperatorTest.adjoint
`OperatorTest.`**`adjoint`**()
Verify that *Operator.adjoint* works appropriately.

References

Wikipedia article on Adjoint.

OperatorTest.derivative
`OperatorTest.`**`derivative`**(*step=0.0001*)
Verify that *`Operator.derivative`* works appropriately.

References

Wikipedia article on Derivative. Wikipedia article on Frechet derivative.

OperatorTest.linear
`OperatorTest.`**`linear`**()
Verify that the operator is actually linear.

OperatorTest.norm
`OperatorTest.`**`norm`**()
Estimate the operator norm of the operator.

The norm is estimated by calculating

```
A(x).norm() / x.norm()
```

for some nonzero `x`

Returnsnorm : `float`

Estimate of operator norm

References

Wikipedia article on Operator norm.

OperatorTest.run_tests
`OperatorTest.`**`run_tests`**()
Run all tests on this operator.

OperatorTest.self_adjoint
`OperatorTest.`**`self_adjoint`**()
Verify (Ax, y) = (x, Ay)
`__init__`(*operator*, *operator_norm=None*)
Create a new instance

Parametersoperator : *`Operator`*

The operator to run tests on

operator_norm : `float`

The norm of the operator, used for error estimates can be estimated otherwise.

space

Standardized tests for *LinearSpace*'s.

Classes

SpaceTest(space[, eps])	Automated tests for *LinearSpace* instances.

SpaceTest

class odl.diagnostics.space.**SpaceTest**(*space*, *eps=1e-05*)
Bases: object

Automated tests for *LinearSpace* instances.

This class allows users to automatically test various features of an LinearSpace such as linearity and the various operators.

Methods

__eq__	Return self==value.
_lincomb()	
contains()	Verify *LinearSpace.__contains__*.
dist()	Verify *LinearSpace.dist*.
element()	Verify *LinearSpace.element*
equals()	Verify LinearSpace.__eq__.
field()	Verify *LinearSpace.field*
inner()	Verify *LinearSpace.inner*.
linearity()	Verifies the linear space properties by examples
multiply()	Verify *LinearSpace.multiply*.
norm()	Verify *LinearSpace.norm*.
run_tests()	Run all tests on this space.
vector()	Verify *LinearSpaceVector*.
vector_assign()	Verify *LinearSpaceVector.assign*.
vector_copy()	Verify *LinearSpaceVector.copy*.
vector_equals()	Verify *LinearSpaceVector.__eq__*.
vector_set_zero()	Verify *LinearSpaceVector.set_zero*.
vector_space()	Verify *LinearSpaceVector.space*.

SpaceTest._lincomb
SpaceTest.**_lincomb**()

SpaceTest.contains
SpaceTest.**contains**()
Verify *LinearSpace.__contains__*.

SpaceTest.dist

SpaceTest.**dist**()
Verify *LinearSpace.dist*.

The dist satisfies properties

positivity `d(x, y) >= 0`

coincidence `d(x, y) = 0` iff `x = y`

symmetry `d(x, y) = d(y, x)`

triangle inequality `d(x, z) = d(x, y) + d(y, z)`

References

Wikipedia article on metric

SpaceTest.element
SpaceTest.**element**()
Verify *LinearSpace.element*

SpaceTest.equals
SpaceTest.**equals**()
Verify `LinearSpace.__eq__`.

SpaceTest.field
SpaceTest.**field**()
Verify *LinearSpace.field*

SpaceTest.inner
SpaceTest.**inner**()
Verify *LinearSpace.inner*.

The inner product satisfies properties such as

conjugate symmetry `(x, y) = (y, x)^*` (^* complex conjugate)

linearity `(a * x, y) = a * (x, y) (x + y, z) = (x, z) + (y, z)`

positivity `(x, x) >= 0`

References

Wikipedia article on inner product.

SpaceTest.linearity
SpaceTest.**linearity**()
Verifies the linear space properties by examples

These properties include things such as associativity

`x + y = y + x`

and identity of the *LinearSpace.zero* element

`x + 0 = x`

References

Wikipedia article on Vector space.

SpaceTest.multiply
`SpaceTest.`**`multiply`**`()`
Verify *`LinearSpace.multiply`*.

Multiplication satisfies

Zero element `0 * x = 0`

Commutativity `x * y = y * x`

Associativity `x * (y * z) = (x * y) * z`

Distributivity `a * (x + y) = a * x + a * y x * (y + z) = x * y + x * z`

SpaceTest.norm
`SpaceTest.`**`norm`**`()`
Verify *`LinearSpace.norm`*.

The norm satisfies properties

linearity `||a * x|| = |a| * ||x||`

triangle inequality `||x + y|| = ||x|| + ||y||`

separation `||x|| = 0` iff `x = 0`

positivity `||x|| >= 0`

References

Wikipedia article on norm.

SpaceTest.run_tests
`SpaceTest.`**`run_tests`**`()`
Run all tests on this space.

SpaceTest.vector
`SpaceTest.`**`vector`**`()`
Verify *`LinearSpaceVector`*.

SpaceTest.vector_assign
`SpaceTest.`**`vector_assign`**`()`
Verify *`LinearSpaceVector.assign`*.

SpaceTest.vector_copy
`SpaceTest.`**`vector_copy`**`()`
Verify *`LinearSpaceVector.copy`*.

SpaceTest.vector_equals
SpaceTest.**vector_equals**()
Verify *LinearSpaceVector.__eq__*.

SpaceTest.vector_set_zero
SpaceTest.**vector_set_zero**()
Verify *LinearSpaceVector.set_zero*.

SpaceTest.vector_space
SpaceTest.**vector_space**()
Verify *LinearSpaceVector.space*.
__init__(*space*, *eps=1e-05*)
Initialize an instance

Parametersspace : *LinearSpace*

The space that should be tested

eps : float, optional

Precision of the tests.

2.4.2 discr

Discretizations in ODL.

Modules

discr_mappings

Classes

FunctionSetMapping(map_type, fset, grid, dspace)	Abstract base class for function set discretization mappings.
GridCollocation(ip_fspace, grid, dspace[, order])	Function evaluation at grid points.
LinearInterpolation(ip_fspace, grid, dspace)	Linear interpolation interpolation as a linear operator.
NearestInterpolation(ip_fspace, grid, dspace)	Nearest neighbor interpolation as a linear operator.
RawGridCollocation(ip_fset, grid, dspace[, ...])	Function evaluation at grid points.
RawNearestInterpolation(ip_fset, grid, dspace)	Nearest neighbor interpolation as a raw *Operator*.

FunctionSetMapping

class odl.discr.discr_mappings.**FunctionSetMapping**(*map_type*, *fset*, *grid*, *dspace*, *order='C'*, *linear=False*)

Bases: *odl.operator.operator.Operator*

Abstract base class for function set discretization mappings.

Attributes

adjoint	The operator adjoint (abstract).
domain	The domain of this operator.
grid	The sampling grid.
inverse	Return the operator inverse.
is_functional	`True` if the this operator's range is a *Field*.
is_linear	`True` if this operator is linear.
order	The axis ordering.
range	The range of this operator.

FunctionSetMapping.adjoint
`FunctionSetMapping.`**`adjoint`**
The operator adjoint (abstract).

RaisesOpNotImplementedError

Since the adjoint cannot be default implemented.

FunctionSetMapping.domain
`FunctionSetMapping.`**`domain`**
The domain of this operator.

FunctionSetMapping.grid
`FunctionSetMapping.`**`grid`**
The sampling grid.

FunctionSetMapping.inverse
`FunctionSetMapping.`**`inverse`**
Return the operator inverse.

RaisesOpNotImplementedError

Since the inverse cannot be default implemented.

FunctionSetMapping.is_functional
`FunctionSetMapping.`**`is_functional`**
`True` if the this operator's range is a *Field*.

FunctionSetMapping.is_linear
`FunctionSetMapping.`**`is_linear`**
`True` if this operator is linear.

FunctionSetMapping.order
`FunctionSetMapping.`**`order`**
The axis ordering.

FunctionSetMapping.range
`FunctionSetMapping.`**`range`**
The range of this operator.

Methods

__eq__(other)	

FunctionSetMapping.__eq__
FunctionSetMapping.**__eq__**(*other*)
__init__(*map_type*, *fset*, *grid*, *dspace*, *order='C'*, *linear=False*)
Initialize a new instance.

Parametersmap_type : {'restriction', 'extension'}

The type of operator

fset : *FunctionSet*

The undiscretized (abstract) set of functions to be discretized

grid : *TensorGrid*

The grid on which to evaluate. Must be contained in the common domain of the function set.

dspace : *NtuplesBase*

Data space providing containers for the values of a discretized object. Its dimension must be equal to the total number of grid points.

order : {'C', 'F'}, optional

Ordering of the values in the flat data arrays. 'C' means the first grid axis varies fastest, the last most slowly, 'F' vice versa.

GridCollocation

class odl.discr.discr_mappings.**GridCollocation**(*ip_fspace*, *grid*, *dspace*, *order='C'*)
Bases: *odl.discr.discr_mappings.RawGridCollocation*

Function evaluation at grid points.

This is the linear *Operator* version of the default 'restriction' used by all core discretization classes.

Attributes

adjoint	The operator adjoint (abstract).
domain	The domain of this operator.
grid	The sampling grid.
inverse	Return the operator inverse.
is_functional	`True` if the this operator's range is a *Field*.
is_linear	`True` if this operator is linear.
order	The axis ordering.
range	The range of this operator.

GridCollocation.adjoint
GridCollocation.**adjoint**
The operator adjoint (abstract).

RaisesOpNotImplementedError

Since the adjoint cannot be default implemented.

GridCollocation.domain
`GridCollocation.`**`domain`**
The domain of this operator.

GridCollocation.grid
`GridCollocation.`**`grid`**
The sampling grid.

GridCollocation.inverse
`GridCollocation.`**`inverse`**
Return the operator inverse.

RaisesOpNotImplementedError

Since the inverse cannot be default implemented.

GridCollocation.is_functional
`GridCollocation.`**`is_functional`**
`True` if the this operator's range is a *`Field`*.

GridCollocation.is_linear
`GridCollocation.`**`is_linear`**
`True` if this operator is linear.

GridCollocation.order
`GridCollocation.`**`order`**
The axis ordering.

GridCollocation.range
`GridCollocation.`**`range`**
The range of this operator.

Methods

`__eq__`(other)	
`_call`(inp)	The raw call method for out-of-place evaluation.

GridCollocation.__eq__
`GridCollocation.`**`__eq__`**(*other*)

GridCollocation._call
`GridCollocation.`**`_call`**(*inp*)
The raw call method for out-of-place evaluation.

Parametersinp : *FunctionSetVector*

The function to be evaluated. It must accept point coordinates in list form (`f(x, y, z)` rather than `f(point)`) and return either a NumPy array of the correct type (defined by the *Ntuples* instance) or a single value.

Returnsoutp : *NtuplesVector*

The function values at the grid points.

See also:

`TensorGrid.meshgrid`, `numpy.meshgrid`

Notes

The code of this call tries to make use of vectorization of the input function, which makes execution much faster and memory-saving. If this fails, it falls back to a slow loop-based variant.

Write your function such that every variable occurs - otherwise, the values will not be broadcasted to the correct size (see example below).

Avoid using the `numpy.vectorize` function - it is merely a convenience function and will not give any speed benefit.

Examples

Define the grid:

```
>>> from odl import TensorGrid
>>> grid = TensorGrid([1, 2], [3, 4, 5], as_midp=True)
```

The `dspace` backend is *Rn*:

```
>>> from odl import Rn
>>> rn = Rn(grid.ntotal)
```

Define a set of functions from the convex hull of the grid to the real numbers:

```
>>> from odl import FunctionSet, RealNumbers
>>> funcset = FunctionSet(grid.convex_hull(), RealNumbers())
```

Finally create the operator:

```
>>> coll_op = RawGridCollocation(funcset, grid, rn)
>>> func_elem = funcset.element(lambda x, y: x - y)
>>> coll_op(func_elem)
Rn(6).element([-2.0, -3.0, -4.0, -1.0, -2.0, -3.0])
```

Or, if we want Fortran ordering:

```
>>> coll_op = RawGridCollocation(funcset, grid, rn, order='F')
>>> coll_op(func_elem)
Rn(6).element([-2.0, -1.0, -3.0, -2.0, -4.0, -3.0])
```

__init__ (*ip_fspace*, *grid*, *dspace*, *order='C'*)

Initialize a new instance.

Parametersfspace : *FunctionSpace*

The undiscretized (abstract) space of functions to be discretized. Its field must be the same as that of data space. Its *Operator.domain* must be an *IntervalProd*.

grid : *TensorGrid*

The grid on which to evaluate. Must be contained in the common domain of the function set.

dspace : *FnBase*

Data space providing containers for the values of a discretized object. Its size must be equal to the total number of grid points. Its field must be the same as that of the function space.

order : {'C', 'F'}, optional

Ordering of the values in the flat data arrays. 'C' means the first grid axis varies fastest, the last most slowly, 'F' vice versa.

LinearInterpolation

class odl.discr.discr_mappings.**LinearInterpolation**(*ip_fspace*, *grid*, *dspace*, *order='C'*)
Bases: *odl.discr.discr_mappings.FunctionSetMapping*

Linear interpolation interpolation as a linear operator.

Attributes

adjoint	The operator adjoint (abstract).
domain	The domain of this operator.
grid	The sampling grid.
inverse	Return the operator inverse.
is_functional	`True` if the this operator's range is a *Field*.
is_linear	`True` if this operator is linear.
order	The axis ordering.
range	The range of this operator.

LinearInterpolation.adjoint
LinearInterpolation.**adjoint**
The operator adjoint (abstract).

RaisesOpNotImplementedError

Since the adjoint cannot be default implemented.

LinearInterpolation.domain
LinearInterpolation.**domain**
The domain of this operator.

LinearInterpolation.grid
LinearInterpolation.**grid**
The sampling grid.

LinearInterpolation.inverse
`LinearInterpolation.`**`inverse`**
Return the operator inverse.

RaisesOpNotImplementedError

Since the inverse cannot be default implemented.

LinearInterpolation.is_functional
`LinearInterpolation.`**`is_functional`**
`True` if the this operator's range is a *`Field`*.

LinearInterpolation.is_linear
`LinearInterpolation.`**`is_linear`**
`True` if this operator is linear.

LinearInterpolation.order
`LinearInterpolation.`**`order`**
The axis ordering.

LinearInterpolation.range
`LinearInterpolation.`**`range`**
The range of this operator.

Methods

`__eq__`(other)	
`_call`(inp)	The raw call method for out-of-place evaluation.

LinearInterpolation.__eq__
`LinearInterpolation.`**`__eq__`**(*other*)

LinearInterpolation._call
`LinearInterpolation.`**`_call`**(*inp*)
The raw call method for out-of-place evaluation.

Parametersinp : *`NtuplesVector`*

The array of numbers to be interpolated

Returnsoutp : *`FunctionSetVector`*

A function (nearest-neighbor) interpolating at a given point or array of points.

Examples

Let's define the complex function space L^2 on a rectangle:

TODO: implement an example!

`__init__`(*ip_fspace*, *grid*, *dspace*, *order='C'*)
Initialize a new *`LinearInterpolation`* instance.

Parametersfspace : *FunctionSpace*

The undiscretized (abstract) space of functions to be discretized. Its field must be the same as that of data space. Its *Operator.domain* must be an *IntervalProd*.

grid : *TensorGrid*

The grid on which to evaluate. Must be contained in the common domain of the function set.

dspace : *FnBase*

Data space providing containers for the values of a discretized object. Its size must be equal to the total number of grid points. Its field must be the same as that of the function space.

order : {'C', 'F'}, optional

Ordering of the values in the flat data arrays. 'C' means the first grid axis varies fastest, the last most slowly, 'F' vice versa.

NearestInterpolation

class odl.discr.discr_mappings.**NearestInterpolation**(*ip_fspace*, *grid*, *dspace*, *order='C'*)

Bases: *odl.discr.discr_mappings.RawNearestInterpolation*

Nearest neighbor interpolation as a linear operator.

Attributes

adjoint	The operator adjoint (abstract).
domain	The domain of this operator.
grid	The sampling grid.
inverse	Return the operator inverse.
is_functional	`True` if the this operator's range is a *Field*.
is_linear	`True` if this operator is linear.
order	The axis ordering.
range	The range of this operator.

NearestInterpolation.adjoint

NearestInterpolation.**adjoint**

The operator adjoint (abstract).

RaisesOpNotImplementedError

Since the adjoint cannot be default implemented.

NearestInterpolation.domain

NearestInterpolation.**domain**

The domain of this operator.

NearestInterpolation.grid

NearestInterpolation.**grid**

The sampling grid.

NearestInterpolation.inverse
`NearestInterpolation.inverse`
Return the operator inverse.

RaisesOpNotImplementedError

Since the inverse cannot be default implemented.

NearestInterpolation.is_functional
`NearestInterpolation.is_functional`
`True` if the this operator's range is a *Field*.

NearestInterpolation.is_linear
`NearestInterpolation.is_linear`
`True` if this operator is linear.

NearestInterpolation.order
`NearestInterpolation.order`
The axis ordering.

NearestInterpolation.range
`NearestInterpolation.range`
The range of this operator.

Methods

__eq__(other)	
_call(inp)	The raw `call` method for out-of-place evaluation.

NearestInterpolation.__eq__
`NearestInterpolation.__eq__`(*other*)

NearestInterpolation._call
`NearestInterpolation._call`(*inp*)
The raw `call` method for out-of-place evaluation.

Parametersinp : *NtuplesVector*

The array of numbers to be interpolated

Returnsoutp : *FunctionSetVector*

A function (nearest-neighbor) interpolating at a given point or array of points.

Examples

Let's define a set of functions from the unit rectangle to one-character strings:

```
>>> from __future__ import unicode_literals, print_function
>>> from odl import Rectangle, Strings
```

```
>>> rect = Rectangle([0, 0], [1, 1])
>>> strings = Strings(1)  # 1-char strings
```

Initialize the space

```
>>> from odl import FunctionSet
>>> space = FunctionSet(rect, strings)
```

The grid is defined by uniform sampling (*TensorGrid.as_midp* indicates that the points will be cell midpoints instead of corners).

```
>>> from odl import uniform_sampling, Ntuples
>>> grid = uniform_sampling(rect, [4, 2], as_midp=True)
>>> grid.coord_vectors
(array([ 0.125,  0.375,  0.625,  0.875]), array([ 0.25,  0.75]))
```

```
>>> dspace = Ntuples(grid.ntotal, dtype='U1')
```

Now initialize the operator:

```
>>> interp_op = RawNearestInterpolation(space, grid, dspace,
...                                     order='C')
```

We test some simple values:

```
>>> import numpy as np
>>> val_arr = np.array([c for c in 'mystring'])
>>> values = dspace.element(val_arr)
>>> function = interp_op(values)
>>> val = function(0.3, 0.6)  # closest to index (1, 1) -> 3
>>> print(val)
t
```

__init__(*ip_fspace*, *grid*, *dspace*, *order='C'*)

Initialize a new *NearestInterpolation* instance.

Parametersfspace : *FunctionSpace*

The undiscretized (abstract) space of functions to be discretized. Its field must be the same as that of data space. Its *Operator.domain* must be an *IntervalProd*.

grid : *TensorGrid*

The grid on which to evaluate. Must be contained in the common domain of the function set.

dspace : *FnBase*

Data space providing containers for the values of a discretized object. Its size must be equal to the total number of grid points. Its field must be the same as that of the function space.

order : {'C', 'F'}, optional

Ordering of the values in the flat data arrays. 'C' means the first grid axis varies fastest, the last most slowly, 'F' vice versa.

Examples

Let's define the complex function space L^2 on a rectangle:

```
>>> from odl import Rectangle, FunctionSpace, ComplexNumbers
>>> rect = Rectangle([0, 0], [1, 1])
>>> space = FunctionSpace(rect, field=ComplexNumbers())
```

The grid is defined by uniform sampling (*TensorGrid.as_midp* indicates that the points will be cell midpoints instead of corners).

```
>>> from odl import uniform_sampling, Cn
>>> grid = uniform_sampling(rect, [4, 2], as_midp=True)
>>> grid.coord_vectors
(array([ 0.125,  0.375,  0.625,  0.875]), array([ 0.25,  0.75]))
>>> dspace = Cn(grid.ntotal)
```

Now initialize the operator:

```
>>> interp_op = RawNearestInterpolation(space, grid, dspace,
...                                     order='C')
```

We test some simple values:

```
>>> import numpy as np
>>> val_arr = np.arange(8) + 1j * np.arange(1, 9)
>>> values = dspace.element(val_arr)
>>> function = interp_op(values)
>>> function(0.3, 0.6)  # closest to index (1, 1) -> 3
(3+4j)
```

RawGridCollocation

class odl.discr.discr_mappings.**RawGridCollocation**(*ip_fset*, *grid*, *dspace*, *order='C'*, *linear=False*)

Bases: *odl.discr.discr_mappings.FunctionSetMapping*

Function evaluation at grid points.

This is the raw *Operator* version of the default 'restriction' used by all core discretization classes.

Attributes

adjoint	The operator adjoint (abstract).
domain	The domain of this operator.
grid	The sampling grid.
inverse	Return the operator inverse.
is_functional	`True` if the this operator's range is a *Field*.
is_linear	`True` if this operator is linear.
order	The axis ordering.
range	The range of this operator.

RawGridCollocation.adjoint

RawGridCollocation.**adjoint**

The operator adjoint (abstract).

RaisesOpNotImplementedError

Since the adjoint cannot be default implemented.

RawGridCollocation.domain
`RawGridCollocation.`**`domain`**
The domain of this operator.

RawGridCollocation.grid
`RawGridCollocation.`**`grid`**
The sampling grid.

RawGridCollocation.inverse
`RawGridCollocation.`**`inverse`**
Return the operator inverse.

RaisesOpNotImplementedError

Since the inverse cannot be default implemented.

RawGridCollocation.is_functional
`RawGridCollocation.`**`is_functional`**
`True` if the this operator's range is a *`Field`*.

RawGridCollocation.is_linear
`RawGridCollocation.`**`is_linear`**
`True` if this operator is linear.

RawGridCollocation.order
`RawGridCollocation.`**`order`**
The axis ordering.

RawGridCollocation.range
`RawGridCollocation.`**`range`**
The range of this operator.

Methods

`__eq__`(other)	
`_call`(inp)	The raw call method for out-of-place evaluation.

RawGridCollocation.__eq__
`RawGridCollocation.`**`__eq__`**`(`*`other`*`)`

RawGridCollocation._call
`RawGridCollocation.`**`_call`**`(`*`inp`*`)`
The raw call method for out-of-place evaluation.

Parametersinp : *`FunctionSetVector`*

The function to be evaluated. It must accept point coordinates in list form (`f(x, y, z)` rather than `f(point)`) and return either a NumPy array of the correct type (defined by the *`Ntuples`* instance) or a single value.

Returnsoutp : *NtuplesVector*

The function values at the grid points.

See also:

`TensorGrid.meshgrid`, `numpy.meshgrid`

Notes

The code of this call tries to make use of vectorization of the input function, which makes execution much faster and memory-saving. If this fails, it falls back to a slow loop-based variant.

Write your function such that every variable occurs - otherwise, the values will not be broadcasted to the correct size (see example below).

Avoid using the `numpy.vectorize` function - it is merely a convenience function and will not give any speed benefit.

Examples

Define the grid:

```
>>> from odl import TensorGrid
>>> grid = TensorGrid([1, 2], [3, 4, 5], as_midp=True)
```

The `dspace` backend is *Rn*:

```
>>> from odl import Rn
>>> rn = Rn(grid.ntotal)
```

Define a set of functions from the convex hull of the grid to the real numbers:

```
>>> from odl import FunctionSet, RealNumbers
>>> funcset = FunctionSet(grid.convex_hull(), RealNumbers())
```

Finally create the operator:

```
>>> coll_op = RawGridCollocation(funcset, grid, rn)
>>> func_elem = funcset.element(lambda x, y: x - y)
>>> coll_op(func_elem)
Rn(6).element([-2.0, -3.0, -4.0, -1.0, -2.0, -3.0])
```

Or, if we want Fortran ordering:

```
>>> coll_op = RawGridCollocation(funcset, grid, rn, order='F')
>>> coll_op(func_elem)
Rn(6).element([-2.0, -1.0, -3.0, -2.0, -4.0, -3.0])
```

__init__(*ip_fset*, *grid*, *dspace*, *order='C'*, *linear=False*)

Initialize a new instance.

Parametersip_fset : *FunctionSet*

The undiscretized (abstract) set of functions to be discretized. The function domain must be an *IntervalProd*.

grid : *TensorGrid*

The grid on which to evaluate. Must be contained in the common domain of the function set.

dspace : *NtuplesBase*

Data space providing containers for the values of a discretized object. Its size must be equal to the total number of grid points.

order : {'C', 'F'}, optional

Ordering of the values in the flat data arrays. 'C' means the first grid axis varies fastest, the last most slowly, 'F' vice versa.

RawNearestInterpolation

class odl.discr.discr_mappings.**RawNearestInterpolation**(*ip_fset*, *grid*, *dspace*, *order='C'*, *linear=False*)

Bases: *odl.discr.discr_mappings.FunctionSetMapping*

Nearest neighbor interpolation as a raw *Operator*.

Attributes

adjoint	The operator adjoint (abstract).
domain	The domain of this operator.
grid	The sampling grid.
inverse	Return the operator inverse.
is_functional	`True` if the this operator's range is a *Field*.
is_linear	`True` if this operator is linear.
order	The axis ordering.
range	The range of this operator.

RawNearestInterpolation.adjoint

RawNearestInterpolation.**adjoint**

The operator adjoint (abstract).

RaisesOpNotImplementedError

Since the adjoint cannot be default implemented.

RawNearestInterpolation.domain

RawNearestInterpolation.**domain**

The domain of this operator.

RawNearestInterpolation.grid

RawNearestInterpolation.**grid**

The sampling grid.

RawNearestInterpolation.inverse

RawNearestInterpolation.**inverse**

Return the operator inverse.

RaisesOpNotImplementedError

Since the inverse cannot be default implemented.

RawNearestInterpolation.is_functional
`RawNearestInterpolation.is_functional`
`True` if the this operator's range is a *Field*.

RawNearestInterpolation.is_linear
`RawNearestInterpolation.is_linear`
`True` if this operator is linear.

RawNearestInterpolation.order
`RawNearestInterpolation.order`
The axis ordering.

RawNearestInterpolation.range
`RawNearestInterpolation.range`
The range of this operator.

Methods

__eq__(other)	
_call(inp)	The raw `call` method for out-of-place evaluation.

RawNearestInterpolation.__eq__
`RawNearestInterpolation.__eq__`(*other*)

RawNearestInterpolation._call
`RawNearestInterpolation._call`(*inp*)
The raw `call` method for out-of-place evaluation.

Parametersinp : *NtuplesVector*

The array of numbers to be interpolated

Returnsoutp : *FunctionSetVector*

A function (nearest-neighbor) interpolating at a given point or array of points.

Examples

Let's define a set of functions from the unit rectangle to one-character strings:

```
>>> from __future__ import unicode_literals, print_function
>>> from odl import Rectangle, Strings
>>> rect = Rectangle([0, 0], [1, 1])
>>> strings = Strings(1)  # 1-char strings
```

Initialize the space

```
>>> from odl import FunctionSet
>>> space = FunctionSet(rect, strings)
```

The grid is defined by uniform sampling (*TensorGrid.as_midp* indicates that the points will be cell midpoints instead of corners).

```
>>> from odl import uniform_sampling, Ntuples
>>> grid = uniform_sampling(rect, [4, 2], as_midp=True)
>>> grid.coord_vectors
(array([ 0.125,  0.375,  0.625,  0.875]), array([ 0.25,  0.75]))
```

```
>>> dspace = Ntuples(grid.ntotal, dtype='U1')
```

Now initialize the operator:

```
>>> interp_op = RawNearestInterpolation(space, grid, dspace,
...                                     order='C')
```

We test some simple values:

```
>>> import numpy as np
>>> val_arr = np.array([c for c in 'mystring'])
>>> values = dspace.element(val_arr)
>>> function = interp_op(values)
>>> val = function(0.3, 0.6)  # closest to index (1, 1) -> 3
>>> print(val)
t
```

__init__(*ip_fset*, *grid*, *dspace*, *order='C'*, *linear=False*)

Initialize a new *RawNearestInterpolation* instance.

Parametersip_fset : *FunctionSet*

The undiscretized (abstract) set of functions to be discretized. The function domain must be an *IntervalProd*.

grid : *TensorGrid*

The grid on which to evaluate. Must be contained in the common domain of the function set.

dspace : *NtuplesBase*

Data space providing containers for the values of a discretized object. Its size must be equal to the total number of grid points.

order : {'C', 'F'}, optional

Ordering of the values in the flat data arrays. 'C' means the first grid axis varies fastest, the last most slowly, 'F' vice versa.

discr_ops

Operators defined on *DiscreteLp*.

Classes

DiscreteDivergence(space[, method])	Divergence operator for *DiscreteLp* spaces.
DiscreteGradient(space[, method])	Spatial gradient operator for *DiscreteLp* spaces.
DiscretePartDeriv(space[, axis, dx, ...])	Calculate the discrete partial derivative along a given axis.

DiscreteDivergence

class odl.discr.discr_ops.**DiscreteDivergence**(*space*, *method='central'*)

Bases: *odl.operator.operator.Operator*

Divergence operator for *DiscreteLp* spaces.

Calls helper function *finite_diff* for each component of the input product space vector. For the adjoint of the *DiscreteDivergence* operator to match the negative *DiscreteGradient* operator implicit zero padding is assumed.

Attributes

adjoint	Return the adjoint operator.
domain	The domain of this operator.
inverse	Return the operator inverse.
is_functional	`True` if the this operator's range is a *Field*.
is_linear	`True` if this operator is linear.
range	The range of this operator.

DiscreteDivergence.adjoint

DiscreteDivergence.**adjoint**

Return the adjoint operator.

Assuming implicit zero padding the adjoint operator is given by the negative of the *DiscreteGradient* operator.

DiscreteDivergence.domain

DiscreteDivergence.**domain**

The domain of this operator.

DiscreteDivergence.inverse

DiscreteDivergence.**inverse**

Return the operator inverse.

RaisesOpNotImplementedError

Since the inverse cannot be default implemented.

DiscreteDivergence.is_functional

DiscreteDivergence.**is_functional**

`True` if the this operator's range is a *Field*.

DiscreteDivergence.is_linear

DiscreteDivergence.**is_linear**

`True` if this operator is linear.

DiscreteDivergence.range

DiscreteDivergence.**range**

The range of this operator.

Methods

`__eq__`	Return self==value.
`_apply`(x, out)	Apply the divergence operator to `x` and store result in `out`.

DiscreteDivergence._apply

`DiscreteDivergence.`**`_apply`**(*x*, *out*)

Apply the divergence operator to `x` and store result in `out`.

Parametersx : `domain` element

`ProductSpaceVector` to which the divergence operator is applied to

out : `range` element

Output vector to which the result is written

Examples

```
>>> from odl import Rectangle, uniform_discr
>>> data = np.array([[0., 1., 2., 3., 4.],
...                  [1., 2., 3., 4., 5.],
...                  [2., 3., 4., 5., 6.]])
>>> discr = uniform_discr([0, 0], [3, 5], data.shape)
>>> div = DiscreteDivergence(discr)
>>> f = div.domain.element([data, data])
>>> div_f = div(f)
>>> print(div_f)
[[1.0, 2.0, 2.5, 3.0, 1.0],
 [2.0, 2.0, 2.0, 2.0, -1.0],
 [1.0, 0.0, -0.5, -1.0, -5.0]]
>>> g = div.range.element(data ** 2)
>>> adj_g = div.adjoint(g)
>>> g.inner(div_f)
-119.0
>>> f.inner(adj_g)
-119.0
```

`__init__`(*space*, *method='central'*)

Initialize a *`DiscreteDivergence`* operator instance.

Zero padding is assumed for the adjoint of the *`DiscreteDivergence`* operator to match negative *`DiscreteGradient`* operator.

Parametersspace : *`DiscreteLp`*

The space of elements which the operator is acting on

method : {'central', 'forward', 'backward'}, optional

What method that should be used.

DiscreteGradient

class `odl.discr.discr_ops.`**`DiscreteGradient`**(*space*, *method='central'*)

Bases: *`odl.operator.operator.Operator`*

Spatial gradient operator for *DiscreteLp* spaces.

Calls helper function *finite_diff* to calculate each component of the resulting product space vector. For the adjoint of the *DiscreteGradient* operator to match the negative *DiscreteDivergence* operator `zero_padding` is assumed.

Attributes

adjoint	Return the adjoint operator.
domain	The domain of this operator.
inverse	Return the operator inverse.
is_functional	`True` if the this operator's range is a *Field*.
is_linear	`True` if this operator is linear.
range	The range of this operator.

DiscreteGradient.adjoint

`DiscreteGradient.`**`adjoint`**

Return the adjoint operator.

Assuming implicit zero padding, the adjoint operator is given by the negative of the *DiscreteDivergence* operator

Note that the `space` argument of the *DiscreteDivergence* operator is not the range but the domain of the *DiscreteGradient* operator.

DiscreteGradient.domain

`DiscreteGradient.`**`domain`**

The domain of this operator.

DiscreteGradient.inverse

`DiscreteGradient.`**`inverse`**

Return the operator inverse.

RaisesOpNotImplementedError

Since the inverse cannot be default implemented.

DiscreteGradient.is_functional

`DiscreteGradient.`**`is_functional`**

`True` if the this operator's range is a *Field*.

DiscreteGradient.is_linear

`DiscreteGradient.`**`is_linear`**

`True` if this operator is linear.

DiscreteGradient.range

`DiscreteGradient.`**`range`**

The range of this operator.

Methods

`__eq__`	Return self==value.
`_apply`(x, out)	Apply the gradient operator to `x` and store result in `out`.

DiscreteGradient._apply

`DiscreteGradient.`**`_apply`**(*x*, *out*)

Apply the gradient operator to `x` and store result in `out`.

Parameters **x** : `domain` element

Input vector to which the *`DiscreteGradient`* operator is applied to

out : `range` element

Output vector to which the result is written

Examples

```
>>> from odl import uniform_discr
>>> data = np.array([[ 0., 1., 2., 3., 4.],
...                  [ 0., 2., 4., 6., 8.]])
>>> discr = uniform_discr([0,0], [2,5], data.shape)
>>> f = discr.element(data)
>>> grad = DiscreteGradient(discr)
>>> grad_f = grad(f)
>>> print(grad_f[0])
[[0.0, 1.0, 2.0, 3.0, 4.0],
 [-0.0, -0.5, -1.0, -1.5, -2.0]]
>>> print(grad_f[1])
[[0.5, 1.0, 1.0, 1.0, -1.5],
 [1.0, 2.0, 2.0, 2.0, -3.0]]
>>> g = grad.range.element((data, data ** 2))
>>> adj_g = grad.adjoint(g)
>>> print(adj_g)
[[-0.5, -3.0, -6.0, -9.0, 0.5],
 [-2.0, -7.5, -15.0, -22.5, 20.0]]
>>> g.inner(grad_f) - f.inner(adj_g)
0.0
```

`__init__`(*space*, *method='central'*)

Initialize a *`DiscreteGradient`* operator instance.

Zero padding is assumed for the adjoint of the *`DiscreteGradient`* operator to match negative *`DiscreteDivergence`* operator.

Parameters **space** : *`DiscreteLp`*

The space of elements which the operator is acting on.

method : {'central', 'forward', 'backward'}, optional

What method that should be used.

DiscretePartDeriv

class `odl.discr.discr_ops.`**`DiscretePartDeriv`**(*space*, *axis=0*, *dx=1.0*, *edge_order=2*, *zero_padding=False*, *method='central'*)

Bases: *`odl.operator.operator.Operator`*

Calculate the discrete partial derivative along a given axis.

Calls helper function *finite_diff* to calculate finite difference. Preserves the shape of the underlying grid.

Attributes

adjoint	Return the adjoint operator.
domain	The domain of this operator.
inverse	Return the operator inverse.
is_functional	`True` if the this operator's range is a *Field*.
is_linear	`True` if this operator is linear.
range	The range of this operator.

DiscretePartDeriv.adjoint
`DiscretePartDeriv.`**`adjoint`**
Return the adjoint operator.

DiscretePartDeriv.domain
`DiscretePartDeriv.`**`domain`**
The domain of this operator.

DiscretePartDeriv.inverse
`DiscretePartDeriv.`**`inverse`**
Return the operator inverse.

RaisesOpNotImplementedError

Since the inverse cannot be default implemented.

DiscretePartDeriv.is_functional
`DiscretePartDeriv.`**`is_functional`**
`True` if the this operator's range is a *Field*.

DiscretePartDeriv.is_linear
`DiscretePartDeriv.`**`is_linear`**
`True` if this operator is linear.

DiscretePartDeriv.range
`DiscretePartDeriv.`**`range`**
The range of this operator.

Methods

`__eq__`	Return self==value.
_apply(x, out)	Apply gradient operator to `x` and store result in `out`.

DiscretePartDeriv._apply

DiscretePartDeriv.**_apply**(*x*, *out*)

Apply gradient operator to `x` and store result in `out`.

Parametersx : `domain` element

Input vector to which the operator is applied to

out : `range` element

Output vector to which the result is written

Examples

```
>>> from odl import uniform_discr
>>> data = np.array([[ 0.,  1.,  2.,  3.,  4.],
...                  [ 0.,  2.,  4.,  6.,  8.]])
>>> discr = uniform_discr([0, 0], [2, 1], data.shape)
>>> par_div = DiscretePartDeriv(discr)
>>> f = par_div.domain.element(data)
>>> par_div_f = par_div(f)
>>> print(par_div_f)
[[0.0, 1.0, 2.0, 3.0, 4.0],
 [0.0, 1.0, 2.0, 3.0, 4.0]]
```

__init__(*space*, *axis=0*, *dx=1.0*, *edge_order=2*, *zero_padding=False*, *method='central'*)

Initialize an operator instance.

Parametersspace : *DiscreteLp*

The space of elements which the operator is acting on

axis : `int`, optional

The axis along which the partial derivative is evaluated. Default: 0

dx : `float`, optional

Scalars specifying the sampling distances in dimension `axis`. Default distance: 1.0

edge_order : {1, 2}, optional

First-order accurate differences can be used at the boundaries if no zero padding is used. Default edge order: 2

zero_padding : `bool`, optional

Implicit zero padding. Assumes values outside the domain of `f` to be zero. Default: `False`

method : {'central', 'forward', 'backward'}, optional

What method that should be used.

Functions

finite_diff(f[, out, axis, dx, edge_order, ...])	Calculate the partial derivative of `f` along a given `axis`.

finite_diff

odl.discr.discr_ops.**finite_diff**(*f*, *out=None*, *axis=0*, *dx=1.0*, *edge_order=2*, *zero_padding=False*, *method='central'*)

Calculate the partial derivative of `f` along a given `axis`.

The partial derivative is computed using second-order accurate central differences in the interior and either first- or second-order accurate one-sides (forward or backward) differences at the boundaries.

Assuming (implicit) zero padding central differences are used on the interior and on endpoints. Otherwise one-sided differences are used. If `zero_padding` is `False` first-order accuracy can be triggered on endpoints with parameter `edge_order`.

The returned array has the same shape as the input array `f`.

Parametersf : array-like

An N-dimensional array

out : array-like or `None`, optional

An N-dimensional array

axis : `int`, optional

The axis along which the partial derivative is evaluated. Default: 0

dx : `float`, optional

Scalar specifying the sample distances in each dimension `axis`. Default distance: 1.0

edge_order : {1, 2}, optional

First-order accurate differences (1) can be used at the boundaries if no zero padding is used. Default edge order: 2

zero_padding : `bool`, optional

Implicit zero padding. Assumes values outside the domain of `f` to be zero. Default: `False`

method : {'central', 'forward', 'backward'}, optional

The method that should be used.

Returnsout : `numpy.ndarray`

Returns an N-dimensional array of the same shape as `f`

Examples

```
>>> f = np.array([ 0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])
>>> finite_diff(f)
array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.])
>>> finite_diff(f, axis=0, dx=1.0, edge_order=2, zero_padding=False)
array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.])
>>> finite_diff(f, dx=0.5)
array([ 2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.])
>>> finite_diff(f, zero_padding=True)
array([ 0.5,  1. ,  1. ,  1. ,  1. ,  1. ,  1. ,  1. ,  1. , -4. ])
>>> finite_diff(f, zero_padding=False, edge_order=1)
array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.])
>>> out = finite_diff(f)
```

```
    >>> out is finite_diff(f)
    False
    >>> out is finite_diff(f, out)
    True
```

discretization

Classes

Discretization(uspace, dspace[, restr, ext])	Abstract class for discretizations of linear vector spaces.
DiscretizationVector(space, data)	Representation of a *Discretization* element.
RawDiscretization(uspace, dspace[, restr, ext])	Abstract raw discretization class.
RawDiscretizationVector(space, ntuple)	Representation of a *RawDiscretization* element.

Discretization

class odl.discr.discretization.**Discretization**(*uspace*, *dspace*, *restr=None*, *ext=None*, ***kwargs*)

Bases: *odl.discr.discretization.RawDiscretization*, *odl.space.base_ntuples.FnBase*

Abstract class for discretizations of linear vector spaces.

This variant of *RawDiscretization* adds linear structure to all its members. The *RawDiscretization.uspace* is a *LinearSpace*, the *RawDiscretization.dspace* for the data representation is an implementation of $\mathbb{F}^n$, where $\mathbb{F}$ is some *Field*, and both *RawDiscretization.restriction* and *RawDiscretization.extension* are linear *Operator*'s.

Attributes

domain	The domain of the continuous space.
dspace	The data space.
dspace_type	Data space type of this discretization.
dtype	The data type of the representation space.
element_type	*DiscretizationVector*
extension	The operator mapping an n-tuple to a *uspace* element.
field	The field of this vector space.
is_cn	If the space represents the set C^n.
is_rn	If the space represents the set R^n.
real_dtype	The corresponding real data type of this space.
restriction	The operator mapping a *uspace* element to an n-tuple.
size	The number of entries per tuple.
uspace	The undiscretized space.

Discretization.domain

Discretization.**domain**

The domain of the continuous space.

Discretization.dspace

Discretization.**dspace**
The data space.

Discretization.dspace_type
Discretization.**dspace_type**
Data space type of this discretization.

Discretization.dtype
Discretization.**dtype**
The data type of the representation space.

Discretization.element_type
Discretization.**element_type**
DiscretizationVector

Discretization.extension
Discretization.**extension**
The operator mapping an n-tuple to a *uspace* element.

Discretization.field
Discretization.**field**
The field of this vector space.

The field is the set of scalars of the space, that is numbers that the vectors in the space can be multiplied with.

Returns
field : *Field*

The underlying field.

Discretization.is_cn
Discretization.**is_cn**
If the space represents the set C^n.

Tuples of complex numbers.

Discretization.is_rn
Discretization.**is_rn**
If the space represents the set R^n.

Tuples of real numbers.

Discretization.real_dtype
Discretization.**real_dtype**
The corresponding real data type of this space.

Discretization.restriction
Discretization.**restriction**
The operator mapping a *uspace* element to an n-tuple.

Discretization.size
`Discretization.`**`size`**
The number of entries per tuple.

Discretization.uspace
`Discretization.`**`uspace`**
The undiscretized space.

Methods

__eq__(other)	Return `self == other`.
_dist(x1, x2)	Raw distance between two vectors.
_divide(x1, x2, out)	Raw pointwise multiplication of two vectors.
_inner(x1, x2)	Raw inner product of two vectors.
_lincomb(a, x1, b, x2, out)	Raw linear combination.
_multiply(x1, x2, out)	Raw pointwise multiplication of two vectors.
_norm(x)	Raw norm of a vector.
element([inp])	Create an element from `inp` or from scratch.
one()	Create a vector of ones.
zero()	Create a vector of zeros.

Discretization.__eq__
`Discretization.`**`__eq__`**(*other*)
Return `self == other`.

Returnsequals : `bool`

`True` if `other` is a *RawDiscretization* instance and all attributes *uspace*, *dspace*, *restriction* and *extension* of `other` and this discretization are equal, `False` otherwise.

Discretization._dist
`Discretization.`**`_dist`**(*x1*, *x2*)
Raw distance between two vectors.

Discretization._divide
`Discretization.`**`_divide`**(*x1*, *x2*, *out*)
Raw pointwise multiplication of two vectors.

Discretization._inner
`Discretization.`**`_inner`**(*x1*, *x2*)
Raw inner product of two vectors.

Discretization._lincomb
`Discretization.`**`_lincomb`**(*a*, *x1*, *b*, *x2*, *out*)
Raw linear combination.

Discretization._multiply
`Discretization.`**`_multiply`**`(x1, x2, out)`
Raw pointwise multiplication of two vectors.

Discretization._norm
`Discretization.`**`_norm`**`(x)`
Raw norm of a vector.

Discretization.element
`Discretization.`**`element`**`(inp=None)`
Create an element from `inp` or from scratch.

Parametersinp : `object`, optional

The input data to create an element from. Must be recognizable by the *`LinearSpace.element`* method of either *`dspace`* or *`uspace`*.

Returnselement : *`RawDiscretizationVector`*

The discretized element, calculated as `dspace.element(inp)` or `restriction(uspace.element(inp))`, tried in this order.

Discretization.one
`Discretization.`**`one`**`()`
Create a vector of ones.

Discretization.zero
`Discretization.`**`zero`**`()`
Create a vector of zeros.
`__init__``(uspace, dspace, restr=None, ext=None, **kwargs)`
Abstract initialization method.

Intended to be called by subclasses for proper type checking and setting of attributes.

Parametersuspace : *`LinearSpace`*

The (abstract) space to be discretized

dspace : *`FnBase`*

Data space providing containers for the values of a discretized object. Its *`FnBase.field`* attribute must be the same as `uspace.field`.

restr : *`Operator`*, linear, optional

Operator mapping a *`RawDiscretization.uspace`* element to a *`RawDiscretization.dspace`* element. Must satisfy `restr.domain == uspace`, `restr.range == dspace`

ext : *`Operator`*, linear, optional

Operator mapping a *`RawDiscretization.dspace`* element to a *`RawDiscretization.uspace`* element. Must satisfy `ext.domain == dspace`, `ext.range == uspace`.

DiscretizationVector

class odl.discr.discretization.**DiscretizationVector**(*space*, *data*)

Bases: *odl.discr.discretization.RawDiscretizationVector*, *odl.space.base_ntuples.FnBaseVector*

Representation of a *Discretization* element.

Attributes

T	The transpose of a vector, the functional given by (.
dtype	type of data storage.
itemsize	The size in bytes on one element of this type.
nbytes	The number of bytes this vector uses in memory.
ndim	Number of dimensions, always 1.
ntuple	Structure for data storage.
shape	Shape of this vector, equals `(size,)`.
size	size of data storage.
space	Space to which this vector.
ufunc	*NtuplesBaseUFuncs*, access to numpy style ufuncs.

DiscretizationVector.T

DiscretizationVector.**T**

The transpose of a vector, the functional given by (. , self)

Returnstranspose : *InnerProductOperator*

Notes

This function is only defined in inner product spaces.

In a complex space, this takes the conjugate transpose of the vector.

Examples

```
>>> from odl import Rn
>>> import numpy as np
>>> rn = Rn(3)
>>> x = rn.element([1, 2, 3])
>>> y = rn.element([2, 1, 3])
>>> x.T(y)
13.0
```

DiscretizationVector.dtype

DiscretizationVector.**dtype**

type of data storage.

DiscretizationVector.itemsize

DiscretizationVector.**itemsize**

The size in bytes on one element of this type.

DiscretizationVector.nbytes
`DiscretizationVector.`**`nbytes`**
The number of bytes this vector uses in memory.

DiscretizationVector.ndim
`DiscretizationVector.`**`ndim`**
Number of dimensions, always 1.

DiscretizationVector.ntuple
`DiscretizationVector.`**`ntuple`**
Structure for data storage.

DiscretizationVector.shape
`DiscretizationVector.`**`shape`**
Shape of this vector, equals `(size,)`.

DiscretizationVector.size
`DiscretizationVector.`**`size`**
size of data storage.

DiscretizationVector.space
`DiscretizationVector.`**`space`**
Space to which this vector.

DiscretizationVector.ufunc
`DiscretizationVector.`**`ufunc`**
`NtuplesBaseUFuncs`, access to numpy style ufuncs.

These are always available, but may or may not be optimized for the specific space in use.

Methods

`__eq__`(other)	Return `vec == other`.
`__getitem__`(indices)	Access values of this vector.
`__setitem__`(indices, values)	Set values of this vector.
`asarray`([out])	Extract the data of this array as a numpy array.
`copy`()	Create an identical (deep) copy of this vector.

DiscretizationVector.__eq__
`DiscretizationVector.`**`__eq__`**(*other*)
Return `vec == other`.

Returnsequals : `bool`

`True` if all entries of `other` are equal to this vector's entries, `False` otherwise.

DiscretizationVector.__getitem__
`DiscretizationVector.`**`__getitem__`**(*indices*)
Access values of this vector.

Parametersindices : `int` or `slice`

The position(s) that should be accessed

Returnsvalues : *NtuplesBaseVector*

The value(s) at the index (indices)

DiscretizationVector.__setitem__

`DiscretizationVector.__setitem__`(*indices*, *values*)

Set values of this vector.

Parametersindices : `int` or `slice`

The position(s) that should be set

values : {scalar, array-like, *NtuplesBaseVector*}

The value(s) that are to be assigned.

If `index` is an `int`, `value` must be single value.

If `index` is a `slice`, `value` must be broadcastable to the size of the slice (same size, shape (1,) or single value).

DiscretizationVector.asarray

`DiscretizationVector.asarray`(*out=None*)

Extract the data of this array as a numpy array.

Parametersout : `numpy.ndarray`, optional

Array in which the result should be written in-place. Has to be contiguous and of the correct dtype.

DiscretizationVector.copy

`DiscretizationVector.copy`()

Create an identical (deep) copy of this vector.

`__init__`(*space*, *data*)

Initialize a new instance.

RawDiscretization

class `odl.discr.discretization.RawDiscretization`(*uspace*, *dspace*, *restr=None*, *ext=None*, ***kwargs*)

Bases: *odl.space.base_ntuples.NtuplesBase*

Abstract raw discretization class.

A discretization in ODL is a way to encode the transition from an arbitrary set to a set of n-tuples explicitly representable in a computer. The most common use case is the discretization of an infinite-dimensional vector space of functions by means of storing coefficients in a finite basis.

The minimal information required to create a discretization is the set to be discretized ("undiscretized space") and a backend for storage and processing of the n-tuples ("data space" or "discretized space").

As additional information, two mappings can be provided. The first one is an explicit way to map an (abstract) element from the source set to an `n`-tuple. This mapping is called **restriction** in ODL. The second one encodes the converse way of mapping an `n`-tuple to an element of the original set. This mapping is called **extension**.

Attributes

domain	The domain of the continuous space.
dspace	The data space.
dspace_type	Data space type of this discretization.
dtype	The data type of the representation space.
element_type	*RawDiscretizationVector*
extension	The operator mapping an n-tuple to a *uspace* element.
restriction	The operator mapping a *uspace* element to an n-tuple.
size	The number of entries per tuple.
uspace	The undiscretized space.

RawDiscretization.domain
RawDiscretization.**domain**
The domain of the continuous space.

RawDiscretization.dspace
RawDiscretization.**dspace**
The data space.

RawDiscretization.dspace_type
RawDiscretization.**dspace_type**
Data space type of this discretization.

RawDiscretization.dtype
RawDiscretization.**dtype**
The data type of the representation space.

RawDiscretization.element_type
RawDiscretization.**element_type**
RawDiscretizationVector

RawDiscretization.extension
RawDiscretization.**extension**
The operator mapping an n-tuple to a *uspace* element.

RawDiscretization.restriction
RawDiscretization.**restriction**
The operator mapping a *uspace* element to an n-tuple.

RawDiscretization.size
RawDiscretization.**size**
The number of entries per tuple.

RawDiscretization.uspace
RawDiscretization.**uspace**
The undiscretized space.

Methods

__eq__(other)	Return `self == other`.
element([inp])	Create an element from `inp` or from scratch.

RawDiscretization.__eq__
`RawDiscretization.__eq__(other)`
Return `self == other`.

Returnsequals : `bool`

`True` if `other` is a *RawDiscretization* instance and all attributes *uspace*, *dspace*, *restriction* and *extension* of `other` and this discretization are equal, `False` otherwise.

RawDiscretization.element
`RawDiscretization.element(inp=None)`
Create an element from `inp` or from scratch.

Parametersinp : `object`, optional

The input data to create an element from. Must be recognizable by the *LinearSpace.element* method of either *dspace* or *uspace*.

Returnselement : *RawDiscretizationVector*

The discretized element, calculated as `dspace.element(inp)` or `restriction(uspace.element(inp))`, tried in this order.

__init__(*uspace*, *dspace*, *restr=None*, *ext=None*, ***kwargs*)
Abstract initialization method.

Intended to be called by subclasses for proper type checking and setting of attributes.

Parametersuspace : *Set*

The undiscretized (abstract) set to be discretized

dspace : *NtuplesBase*

Data space providing containers for the values of a discretized object

restr : *Operator*, optional

Operator mapping a *uspace* element to a *dspace* element. Must satisfy `restr.domain == uspace`, `restr.range == dspace`.

ext : *Operator*, optional

Operator mapping a *dspace* element to a *uspace* element. Must satisfy `ext.domain == dspace`, `ext.range == uspace`.

RawDiscretizationVector

class `odl.discr.discretization.RawDiscretizationVector(space, ntuple)`
Bases: *odl.space.base_ntuples.NtuplesBaseVector*

Representation of a *RawDiscretization* element.

Basically only a wrapper class for dspace's vector class.

Attributes

dtype	type of data storage.
itemsize	The size in bytes on one element of this type.
nbytes	The number of bytes this vector uses in memory.
ndim	Number of dimensions, always 1.
ntuple	Structure for data storage.
shape	Shape of this vector, equals `(size,)`.
size	size of data storage.
space	Space to which this vector.
ufunc	*NtuplesBaseUFuncs*, access to numpy style ufuncs.

RawDiscretizationVector.dtype
`RawDiscretizationVector.`**`dtype`**
type of data storage.

RawDiscretizationVector.itemsize
`RawDiscretizationVector.`**`itemsize`**
The size in bytes on one element of this type.

RawDiscretizationVector.nbytes
`RawDiscretizationVector.`**`nbytes`**
The number of bytes this vector uses in memory.

RawDiscretizationVector.ndim
`RawDiscretizationVector.`**`ndim`**
Number of dimensions, always 1.

RawDiscretizationVector.ntuple
`RawDiscretizationVector.`**`ntuple`**
Structure for data storage.

RawDiscretizationVector.shape
`RawDiscretizationVector.`**`shape`**
Shape of this vector, equals `(size,)`.

RawDiscretizationVector.size
`RawDiscretizationVector.`**`size`**
size of data storage.

RawDiscretizationVector.space
`RawDiscretizationVector.`**`space`**
Space to which this vector.

RawDiscretizationVector.ufunc
`RawDiscretizationVector.`**`ufunc`**
NtuplesBaseUFuncs, access to numpy style ufuncs.

These are always available, but may or may not be optimized for the specific space in use.

Methods

__eq__(other)	Return `vec == other`.
__getitem__(indices)	Access values of this vector.
__setitem__(indices, values)	Set values of this vector.
asarray([out])	Extract the data of this array as a numpy array.
copy()	Create an identical (deep) copy of this vector.

RawDiscretizationVector.__eq__
`RawDiscretizationVector.`**`__eq__`**(*other*)
Return `vec == other`.

Returnsequals : `bool`

`True` if all entries of `other` are equal to this vector's entries, `False` otherwise.

RawDiscretizationVector.__getitem__
`RawDiscretizationVector.`**`__getitem__`**(*indices*)
Access values of this vector.

Parametersindices : `int` or `slice`

The position(s) that should be accessed

Returnsvalues : *NtuplesBaseVector*

The value(s) at the index (indices)

RawDiscretizationVector.__setitem__
`RawDiscretizationVector.`**`__setitem__`**(*indices*, *values*)
Set values of this vector.

Parametersindices : `int` or `slice`

The position(s) that should be set

values : {scalar, array-like, *NtuplesBaseVector*}

The value(s) that are to be assigned.

If `index` is an `int`, `value` must be single value.

If `index` is a `slice`, `value` must be broadcastable to the size of the slice (same size, shape (1,) or single value).

RawDiscretizationVector.asarray
`RawDiscretizationVector.`**`asarray`**(*out=None*)
Extract the data of this array as a numpy array.

Parametersout : `numpy.ndarray`, optional

Array in which the result should be written in-place. Has to be contiguous and of the correct dtype.

RawDiscretizationVector.copy

`RawDiscretizationVector.`**`copy`**`()`

Create an identical (deep) copy of this vector.

`__init__`(*space*, *ntuple*)

Initialize a new instance.

Functions

dspace_type(space, impl[, dtype])	Select the correct corresponding n-tuples space.

dspace_type

`odl.discr.discretization.`**`dspace_type`**(*space*, *impl*, *dtype=None*)

Select the correct corresponding n-tuples space.

Parametersspace : `object`

The template space. If it has a `field` attribute, `dtype` must be consistent with it

impl : {'numpy', 'cuda'}

The backend for the data space

dtype : `type`, optional

Data type which the space is supposed to use. If `None`, the space type is purely determined from `space` and `impl`. If given, it must be compatible with the field of `space`. Non-floating types result in basic *Fn*-type spaces.

Returnsstype : `type`

Space type selected after the space's field, the backend and the data type

grid

Sparse implementations of n-dimensional sampling grids.

Sampling grids are collections of points in an n-dimensional coordinate space with a certain structure which is exploited to minimize storage.

Classes

RegularGrid(min_pt, max_pt, shape, **kwargs)	An n-dimensional tensor grid with equidistant coordinates.
TensorGrid(*coord_vectors, **kwargs)	An n-dimensional tensor grid.

RegularGrid

class `odl.discr.grid.`**`RegularGrid`**(*min_pt*, *max_pt*, *shape*, ***kwargs*)

Bases: *odl.discr.grid.TensorGrid*

An n-dimensional tensor grid with equidistant coordinates.

This is a sparse representation of an n-dimensional grid defined as the tensor product of n coordinate vectors with equidistant nodes. The grid points are calculated according to the rule

x_j = center + (j - (shape-1)/2 * stride)

with elementwise addition and multiplication. Note that the division is a true division (no rounding), thus if there is an axis with an even number of nodes, the center is not a grid point.

Attributes

as_midp	If the grid is "pixel" centered.
cell_volume	Cell volume of an underlying regular grid.
center	The center of the grid.
coord_vectors	The coordinate vectors of the grid.
max_pt	Vector containing the maximal grid coordinates per axis.
min_pt	Vector containing the minimal grid coordinates per axis.
ndim	The number of dimensions of the grid.
ntotal	The total number of grid points.
order	Axis ordering of this grid.
shape	The number of grid points per axis.
stride	The step per axis between two neighboring grid points.

RegularGrid.as_midp

RegularGrid.**as_midp**

If the grid is "pixel" centered.

RegularGrid.cell_volume

RegularGrid.**cell_volume**

Cell volume of an underlying regular grid.

Examples

```
>>> rg = RegularGrid([0, 0], [1, 1], (2, 2))
>>> rg.cell_volume
1.0
```

RegularGrid.center

RegularGrid.**center**

The center of the grid. Not necessarily a grid point.

Examples

```
>>> rg = RegularGrid([-1.5, -1], [-0.5, 3], (2, 3))
>>> rg.center
array([-1.,  1.])
```

RegularGrid.coord_vectors
RegularGrid.**coord_vectors**
The coordinate vectors of the grid.

RegularGrid.max_pt
RegularGrid.**max_pt**
Vector containing the maximal grid coordinates per axis.

Examples

```
>>> g = TensorGrid([1, 2, 5], [-2, 1.5, 2])
>>> g.max_pt
array([ 5.,  2.])
```

RegularGrid.min_pt
RegularGrid.**min_pt**
Vector containing the minimal grid coordinates per axis.

Examples

```
>>> g = TensorGrid([1, 2, 5], [-2, 1.5, 2])
>>> g.min_pt
array([ 1., -2.])
```

RegularGrid.ndim
RegularGrid.**ndim**
The number of dimensions of the grid.

RegularGrid.ntotal
RegularGrid.**ntotal**
The total number of grid points.

RegularGrid.order
RegularGrid.**order**
Axis ordering of this grid.

RegularGrid.shape
RegularGrid.**shape**
The number of grid points per axis.

RegularGrid.stride
RegularGrid.**stride**
The step per axis between two neighboring grid points.

Examples

```
>>> rg = RegularGrid([-1.5, -1], [-0.5, 3], (2, 3))
>>> rg.stride
array([ 1.,  2.])
```

Methods

__contains__(other)	Return `other in self`.
__eq__(other)	Return `self == other`.
__getitem__(slc)	self[slc] implementation.
approx_contains(other, tol)	Test if `other` belongs to this grid up to a tolerance.
approx_equals(other, tol)	Test if this grid is equal to another grid.
cell_sizes()	The grid cell sizes as coordinate vectors.
convex_hull()	The "inner" of the grid, an *IntervalProd*.
corners([order])	The corner points of the grid in a single array.
element()	An arbitrary element, the minimum coordinates.
insert(grid, index)	Insert another regular grid before the given index.
is_subgrid(other[, tol])	Test if this grid is contained in another grid.
max()	Return vector with maximal cell coordinates per axis.
meshgrid()	A grid suitable for function evaluation.
min()	Return vector with minimal cell coordinates per axis.
points([order])	All grid points in a single array.
size()	Return a vector containing the total sizes.
squeeze()	Remove the degenerate dimensions.

RegularGrid.__contains__

`RegularGrid.__contains__`(*other*)

Return `other in self`.

RegularGrid.__eq__

`RegularGrid.__eq__`(*other*)

Return `self == other`.

RegularGrid.__getitem__

`RegularGrid.__getitem__`(*slc*)

self[slc] implementation.

Parameters

slc : `int` or slice

Negative indices and `None` (new axis) are not supported.

Examples

```
>>> g = RegularGrid([-1.5, -3, -1], [-0.5, 7, 4], (2, 3, 6))
>>> g[0, 0, 0]
array([-1.5, -3. , -1. ])
>>> g[:, 0, 0]
RegularGrid([-1.5, -3.0, -1.0], [-0.5, -3.0, -1.0], [2, 1, 1])
```

Ellipsis can be used, too:

```
>>> g[..., ::3]
RegularGrid([-1.5, -3.0, -1.0], [-0.5, 7.0, 2.0], [2, 3, 2])
```

RegularGrid.approx_contains

`RegularGrid.`**`approx_contains`**(*other*, *tol*)

Test if `other` belongs to this grid up to a tolerance.

Parametersother : array-like or `float`

The object to test for membership in this grid

tol : `float`

Allow deviations up to this number in absolute value per vector entry.

Examples

```
>>> g = TensorGrid([0, 1], [-1, 0, 2])
>>> g.approx_contains([0, 0], tol=0.0)
True
>>> [0, 0] in g  # equivalent
True
>>> g.approx_contains([0.1, -0.1], tol=0.0)
False
>>> g.approx_contains([0.1, -0.1], tol=0.15)
True
```

RegularGrid.approx_equals

`RegularGrid.`**`approx_equals`**(*other*, *tol*)

Test if this grid is equal to another grid.

Parametersother : `object`

Object to be tested

tol : `float`

Allow deviations up to this number in absolute value per vector entry.

Returnsequals : `bool`

`True` if `other` is a *`TensorGrid`* instance with all coordinate vectors equal (up to the given tolerance), to the ones of this grid, otherwise `False`.

Examples

```
>>> g1 = TensorGrid([0, 1], [-1, 0, 2])
>>> g2 = TensorGrid([-0.1, 1.1], [-1, 0.1, 2])
>>> g1.approx_equals(g2, tol=0)
False
>>> g1.approx_equals(g2, tol=0.15)
True
```

RegularGrid.cell_sizes

RegularGrid.**cell_sizes**()

The grid cell sizes as coordinate vectors.

Returnscsizes : tuple of numpy.ndarray

The cell sizes per axis. The length of the vectors will be one less than *coord_vectors* if *as_midp* is False, otherwise they will have the same length. For axes with 1 grid point, cell size is set to 0.

Examples

```
>>> g = TensorGrid([0, 1], [-1, 0, 2])
>>> g.cell_sizes()
(array([ 1.]), array([ 1.,  2.]))
>>> g = TensorGrid([0, 1], [-1, 0, 2], as_midp=True)
>>> g.cell_sizes()
(array([ 1.,  1.]), array([ 1. ,  1.5,  2. ]))
```

RegularGrid.convex_hull

RegularGrid.**convex_hull**()

The "inner" of the grid, an *IntervalProd*.

The convex hull of a set is the union of all line segments between points in the set. For a tensor grid, it is the interval product given by the extremal coordinates.

Returnschull : *IntervalProd*

Interval product defined by the minimum and maximum of the grid (depends on *as_midp*)

Examples

```
>>> g = TensorGrid([-1, 0, 3], [2, 4], [5], [2, 4, 7])
>>> g.convex_hull()
IntervalProd([-1.0, 2.0, 5.0, 2.0], [3.0, 4.0, 5.0, 7.0])
>>> g = TensorGrid([-1, 0, 3], [2, 4], [5], [2, 4, 7],
...                as_midp=True)
>>> g.convex_hull()
IntervalProd([-1.5, 1.0, 5.0, 1.0], [4.5, 5.0, 5.0, 8.5])
```

RegularGrid.corners

RegularGrid.**corners**(*order=None*)

The corner points of the grid in a single array.

Parametersorder : {'C', 'F'}

Force this axis ordering instead of the grid's own

Returnsout : numpy.ndarray

The size of the array is 2^m x ndim, where m is the number of non-degenerate axes, i.e. the corners are stored as rows.

Examples

```
>>> g = TensorGrid([0, 1], [-1, 0, 2])  # default 'C' ordering
>>> g.corners()
array([[ 0., -1.],
       [ 0.,  2.],
       [ 1., -1.],
       [ 1.,  2.]])
>>> g.corners(order='F')
array([[ 0., -1.],
       [ 1., -1.],
       [ 0.,  2.],
       [ 1.,  2.]])
```

RegularGrid.element

`RegularGrid.`**`element`**`()`

An arbitrary element, the minimum coordinates.

RegularGrid.insert

`RegularGrid.`**`insert`**`(`*grid*, *index*`)`

Insert another regular grid before the given index.

The given grid (`m` dimensions) is inserted into the current one (`n` dimensions) before the given index, resulting in a new *`RegularGrid`* with `n + m` dimensions. Note that no changes are made in-place.

Parametersgrid : *`RegularGrid`*

The grid to be inserted.

index : `numbers.Integral`

The index of the dimension before which 'other' is to be inserted. Must fulfill `0 <= index <= ndim`.

Returnsnewgrid : *`RegularGrid`*

The enlarged grid

Examples

```
>>> rg1 = RegularGrid([-1.5, -1], [-0.5, 3], (2, 3))
>>> rg2 = RegularGrid(-3, 7, 6)
>>> rg1.insert(rg2, 1)
RegularGrid([-1.5, -3.0, -1.0], [-0.5, 7.0, 3.0], [2, 6, 3])
```

RegularGrid.is_subgrid

`RegularGrid.`**`is_subgrid`**`(`*other*, *tol=0.0*`)`

Test if this grid is contained in another grid.

Parameterstol : `float`

Allow deviations up to this number in absolute value per coordinate vector entry.

Examples

```
>>> rg = RegularGrid([-2, -2], [0, 4], (3, 4))
>>> rg.coord_vectors
(array([-2., -1.,  0.]), array([-2.,  0.,  2.,  4.]))
>>> rg_sub = RegularGrid([-1, 2], [0, 4], (2, 2))
>>> rg_sub.coord_vectors
(array([-1.,  0.]), array([ 2.,  4.]))
>>> rg_sub.is_subgrid(rg)
True
```

Fuzzy check is also possible. Note that the tolerance still applies to the coordinate vectors.

```
>>> rg_sub = RegularGrid([-1.015, 2], [0, 3.99], (2, 2))
>>> rg_sub.is_subgrid(rg, tol=0.01)
False
>>> rg_sub.is_subgrid(rg, tol=0.02)
True
```

RegularGrid.max

`RegularGrid.`**`max`**`()`

Return vector with maximal cell coordinates per axis.

This is relevant if the grid was initialized with *as_midp* `True`, in which case the maximum is half a cell larger than the maximum grid point.

Examples

```
>>> g = TensorGrid([1, 2, 5], [-2, 1.5, 2], as_midp=False)
>>> g.max()
array([ 5.,  2.])
>>> g = TensorGrid([1, 2, 5], [-2, 1.5, 2], as_midp=True)
>>> g.max()
array([ 6.5 ,  2.25])
```

RegularGrid.meshgrid

`RegularGrid.`**`meshgrid`**`()`

A grid suitable for function evaluation.

Returnsmeshgrid : `tuple` of `numpy.ndarray`

Function evaluation grid with *ndim* axes

See also:

`numpy.meshgrid`coordinate matrices from coordinate vectors We use `indexing='ij'` and `copy=True`

Examples

```
>>> g = TensorGrid([0, 1], [-1, 0, 2])
>>> x, y = g.meshgrid()
>>> x
array([[ 0.],
```

```
       [ 1.]])
>>> y
array([[-1.,  0.,  2.]])
```

Easy function evaluation via broadcasting:

```
>>> x**2 - y**2
array([[-1.,  0., -4.],
       [ 0.,  1., -3.]])
```

Fortran ordering of the grid is respected:

```
>>> g = TensorGrid([0, 1], [-1, 0, 2], order='F')
>>> x, y = g.meshgrid()
>>> x.flags.f_contiguous, y.flags.f_contiguous
(True, True)
```

RegularGrid.min

`RegularGrid.`**`min`**`()`

Return vector with minimal cell coordinates per axis.

This is relevant if the grid was initialized with *as_midp* `True`, in which case the minimum is half a cell smaller than the minimum grid point.

Examples

```
>>> g = TensorGrid([1, 2, 5], [-2, 1.5, 2], as_midp=False)
>>> g.min()
array([ 1., -2.])
>>> g = TensorGrid([1, 2, 5], [-2, 1.5, 2], as_midp=True)
>>> g.min()
array([ 0.5 , -3.75])
```

RegularGrid.points

`RegularGrid.`**`points`**`(`*order=None*`)`

All grid points in a single array.

Parametersorder : {'C', 'F'}

Force this axis ordering instead of the grid's own

Returnspoints : `numpy.ndarray`

The size of the array is ntotal x ndim, i.e. the points are stored as rows.

Examples

```
>>> g = TensorGrid([0, 1], [-1, 0, 2])  # default 'C' ordering
>>> g.points()
array([[ 0., -1.],
       [ 0.,  0.],
       [ 0.,  2.],
       [ 1., -1.],
       [ 1.,  0.],
       [ 1.,  2.]])
```

```
>>> g.points(order='F')
array([[ 0., -1.],
       [ 1., -1.],
       [ 0.,  0.],
       [ 1.,  0.],
       [ 0.,  2.],
       [ 1.,  2.]])
```

RegularGrid.size

RegularGrid.**size**()

Return a vector containing the total sizes.

RegularGrid.squeeze

RegularGrid.**squeeze**()

Remove the degenerate dimensions.

Note that no changes are made in-place.

Returnssqueezed : *RegularGrid*

The squeezed grid

Examples

```
>>> g = RegularGrid([0, 0, 0], [1, 0, 1], (5, 1, 5))
>>> g.squeeze()
RegularGrid([0.0, 0.0], [1.0, 1.0], [5, 5])
```

__init__(*min_pt*, *max_pt*, *shape*, ***kwargs*)

Initialize a new instance.

Parametersmin_pt : array-like or `float`

Grid point with minimum coordinates, can be a single float for 1D grids

max_pt : array-like or `float`

Grid point with maximum coordinates, can be a single float for 1D grids

shape : array-like or `int`

The number of grid points per axis, can be an integer for 1D grids

kwargs : {'as_midp'}

'as_midp'[`bool`, optional] Treat grid points as midpoints of rectangular cells. This influences the behavior of *TensorGrid.min*, *TensorGrid.max* and *TensorGrid.cell_sizes*.

Default: `False`

Examples

```
>>> rg = RegularGrid([-1.5, -1], [-0.5, 3], (2, 3))
>>> rg
RegularGrid([-1.5, -1.0], [-0.5, 3.0], [2, 3])
>>> rg.coord_vectors
```

```
        (array([-1.5, -0.5]), array([-1.,  1.,  3.]))
        >>> rg.ndim, rg.ntotal
        (2, 6)
```

TensorGrid

class odl.discr.grid.**TensorGrid**(**coord_vectors*, ***kwargs*)
Bases: *odl.set.sets.Set*

An n-dimensional tensor grid.

This is a sparse representation of a collection of n-dimensional points defined by the tensor product of n coordinate vectors.

Example: x = (x_1, x_2), y = (y_1, y_2, y_3, y_4), z = (z_1, z_2, z_3). The resulting grid consists of all possible combinations p = (x_i, y_j, z_k), hence 2 * 4 * 3 = 24 points in total.

Attributes

as_midp	If the grid is "pixel" centered.
coord_vectors	The coordinate vectors of the grid.
max_pt	Vector containing the maximal grid coordinates per axis.
min_pt	Vector containing the minimal grid coordinates per axis.
ndim	The number of dimensions of the grid.
ntotal	The total number of grid points.
order	Axis ordering of this grid.
shape	The number of grid points per axis.

TensorGrid.as_midp
TensorGrid.**as_midp**
If the grid is "pixel" centered.

TensorGrid.coord_vectors
TensorGrid.**coord_vectors**
The coordinate vectors of the grid.

TensorGrid.max_pt
TensorGrid.**max_pt**
Vector containing the maximal grid coordinates per axis.

Examples

```
        >>> g = TensorGrid([1, 2, 5], [-2, 1.5, 2])
        >>> g.max_pt
        array([ 5.,  2.])
```

TensorGrid.min_pt
TensorGrid.**min_pt**
Vector containing the minimal grid coordinates per axis.

Examples

```
>>> g = TensorGrid([1, 2, 5], [-2, 1.5, 2])
>>> g.min_pt
array([ 1., -2.])
```

TensorGrid.ndim

`TensorGrid.`**`ndim`**

The number of dimensions of the grid.

TensorGrid.ntotal

`TensorGrid.`**`ntotal`**

The total number of grid points.

TensorGrid.order

`TensorGrid.`**`order`**

Axis ordering of this grid.

TensorGrid.shape

`TensorGrid.`**`shape`**

The number of grid points per axis.

Methods

__contains__(other)	Return `other in self`.
__eq__(other)	Return `self == other`.
__getitem__(slc)	self[slc] implementation.
approx_contains(other, tol)	Test if `other` belongs to this grid up to a tolerance.
approx_equals(other, tol)	Test if this grid is equal to another grid.
cell_sizes()	The grid cell sizes as coordinate vectors.
convex_hull()	The "inner" of the grid, an *IntervalProd*.
corners([order])	The corner points of the grid in a single array.
element()	An arbitrary element, the minimum coordinates.
insert(other, index)	Insert another grid before the given index.
is_subgrid(other[, tol])	Test if this grid is contained in another grid.
max()	Return vector with maximal cell coordinates per axis.
meshgrid()	A grid suitable for function evaluation.
min()	Return vector with minimal cell coordinates per axis.
points([order])	All grid points in a single array.
size()	Return a vector containing the total sizes.
squeeze()	Remove the degenerate dimensions.

TensorGrid.__contains__

`TensorGrid.`**`__contains__`**(*other*)

Return `other in self`.

TensorGrid.__eq__

TensorGrid.__eq__(*other*)
Return `self == other`.

TensorGrid.__getitem__
TensorGrid.__getitem__(*slc*)
self[slc] implementation.

Parameterslc : `int` or slice

Negative indices and `None` (new axis) are not supported.

Examples

```
>>> g = TensorGrid([-1, 0, 3], [2, 4], [5], [2, 4, 7])
>>> g[0, 0, 0, 0]
array([-1.,  2.,  5.,  2.])
>>> g[:, 0, 0, 0]
TensorGrid([-1.0, 0.0, 3.0], [2.0], [5.0], [2.0])
>>> g[0, ..., 1:]
TensorGrid([-1.0], [2.0, 4.0], [5.0], [4.0, 7.0])
>>> g[::2, ..., ::2]
TensorGrid([-1.0, 3.0], [2.0, 4.0], [5.0], [2.0, 7.0])
```

TensorGrid.approx_contains
TensorGrid.approx_contains(*other*, *tol*)
Test if `other` belongs to this grid up to a tolerance.

Parametersother : array-like or `float`

The object to test for membership in this grid

tol : `float`

Allow deviations up to this number in absolute value per vector entry.

Examples

```
>>> g = TensorGrid([0, 1], [-1, 0, 2])
>>> g.approx_contains([0, 0], tol=0.0)
True
>>> [0, 0] in g  # equivalent
True
>>> g.approx_contains([0.1, -0.1], tol=0.0)
False
>>> g.approx_contains([0.1, -0.1], tol=0.15)
True
```

TensorGrid.approx_equals
TensorGrid.approx_equals(*other*, *tol*)
Test if this grid is equal to another grid.

Parametersother : `object`

Object to be tested

tol : `float`

Allow deviations up to this number in absolute value per vector entry.

Returnsequals : `bool`

`True` if `other` is a *TensorGrid* instance with all coordinate vectors equal (up to the given tolerance), to the ones of this grid, otherwise `False`.

Examples

```
>>> g1 = TensorGrid([0, 1], [-1, 0, 2])
>>> g2 = TensorGrid([-0.1, 1.1], [-1, 0.1, 2])
>>> g1.approx_equals(g2, tol=0)
False
>>> g1.approx_equals(g2, tol=0.15)
True
```

TensorGrid.cell_sizes

`TensorGrid.`**`cell_sizes`**`()`

The grid cell sizes as coordinate vectors.

Returnscsizes : `tuple` of `numpy.ndarray`

The cell sizes per axis. The length of the vectors will be one less than *coord_vectors* if *as_midp* is `False`, otherwise they will have the same length. For axes with 1 grid point, cell size is set to 0.

Examples

```
>>> g = TensorGrid([0, 1], [-1, 0, 2])
>>> g.cell_sizes()
(array([ 1.]), array([ 1.,  2.]))
>>> g = TensorGrid([0, 1], [-1, 0, 2], as_midp=True)
>>> g.cell_sizes()
(array([ 1.,  1.]), array([ 1. ,  1.5,  2. ]))
```

TensorGrid.convex_hull

`TensorGrid.`**`convex_hull`**`()`

The "inner" of the grid, an *IntervalProd*.

The convex hull of a set is the union of all line segments between points in the set. For a tensor grid, it is the interval product given by the extremal coordinates.

Returnschull : *IntervalProd*

Interval product defined by the minimum and maximum of the grid (depends on *as_midp*)

Examples

```
>>> g = TensorGrid([-1, 0, 3], [2, 4], [5], [2, 4, 7])
>>> g.convex_hull()
IntervalProd([-1.0, 2.0, 5.0, 2.0], [3.0, 4.0, 5.0, 7.0])
>>> g = TensorGrid([-1, 0, 3], [2, 4], [5], [2, 4, 7],
...                as_midp=True)
>>> g.convex_hull()
IntervalProd([-1.5, 1.0, 5.0, 1.0], [4.5, 5.0, 5.0, 8.5])
```

TensorGrid.corners

`TensorGrid.`**`corners`**(*order=None*)

The corner points of the grid in a single array.

Parametersorder : {'C', 'F'}

Force this axis ordering instead of the grid's own

Returnsout : `numpy.ndarray`

The size of the array is 2^m x ndim, where m is the number of non-degenerate axes, i.e. the corners are stored as rows.

Examples

```
>>> g = TensorGrid([0, 1], [-1, 0, 2])  # default 'C' ordering
>>> g.corners()
array([[ 0., -1.],
       [ 0.,  2.],
       [ 1., -1.],
       [ 1.,  2.]])
>>> g.corners(order='F')
array([[ 0., -1.],
       [ 1., -1.],
       [ 0.,  2.],
       [ 1.,  2.]])
```

TensorGrid.element

`TensorGrid.`**`element`**()

An arbitrary element, the minimum coordinates.

TensorGrid.insert

`TensorGrid.`**`insert`**(*other*, *index*)

Insert another grid before the given index.

The given grid (`m` dimensions) is inserted into the current one (`n` dimensions) before the given index, resulting in a new *`TensorGrid`* with `n + m` dimensions. Note that no changes are made in-place.

Parametersother : *`TensorGrid`*, `float` or array-like

The grid to be inserted. A `float` or array `a` is treated as `TensorGrid(a)`.

index : `numbers.Integral`

The index of the dimension before which `other` is to be inserted. Must fulfill `0 <= index <= ndim`.

Returnsnewgrid : *`TensorGrid`*

The enlarged grid

Examples

```
>>> g1 = TensorGrid([0, 1], [-1, 2])
>>> g2 = TensorGrid([1], [-6, 15])
>>> g1.insert(g2, 1)
TensorGrid([0.0, 1.0], [1.0], [-6.0, 15.0], [-1.0, 2.0])
```

TensorGrid.is_subgrid

`TensorGrid.`**`is_subgrid`**(*other*, *tol=0.0*)

Test if this grid is contained in another grid.

Parameterother : *TensorGrid*

The other grid which is supposed to contain this grid

tol : `float`

Allow deviations up to this number in absolute value per coordinate vector entry.

Examples

```
>>> g = TensorGrid([0, 1], [-1, 0, 2])
>>> g_sub = TensorGrid([0], [-1, 2])
>>> g_sub.is_subgrid(g)
True
>>> g_sub = TensorGrid([0.1], [-1.05, 2.1])
>>> g_sub.is_subgrid(g)
False
>>> g_sub.is_subgrid(g, tol=0.15)
True
```

TensorGrid.max

`TensorGrid.`**`max`**()

Return vector with maximal cell coordinates per axis.

This is relevant if the grid was initialized with *as_midp* `True`, in which case the maximum is half a cell larger than the maximum grid point.

Examples

```
>>> g = TensorGrid([1, 2, 5], [-2, 1.5, 2], as_midp=False)
>>> g.max()
array([ 5.,  2.])
>>> g = TensorGrid([1, 2, 5], [-2, 1.5, 2], as_midp=True)
>>> g.max()
array([ 6.5 ,  2.25])
```

TensorGrid.meshgrid

`TensorGrid.meshgrid()`

A grid suitable for function evaluation.

Returnsmeshgrid : `tuple` of `numpy.ndarray`

Function evaluation grid with *ndim* axes

See also:

numpy.meshgridcoordinate matrices from coordinate vectors We use `indexing='ij'` and `copy=True`

Examples

```
>>> g = TensorGrid([0, 1], [-1, 0, 2])
>>> x, y = g.meshgrid()
>>> x
array([[ 0.],
       [ 1.]])
>>> y
array([[-1.,  0.,  2.]])
```

Easy function evaluation via broadcasting:

```
>>> x**2 - y**2
array([[-1.,  0., -4.],
       [ 0.,  1., -3.]])
```

Fortran ordering of the grid is respected:

```
>>> g = TensorGrid([0, 1], [-1, 0, 2], order='F')
>>> x, y = g.meshgrid()
>>> x.flags.f_contiguous, y.flags.f_contiguous
(True, True)
```

TensorGrid.min

`TensorGrid.min()`

Return vector with minimal cell coordinates per axis.

This is relevant if the grid was initialized with *as_midp* `True`, in which case the minimum is half a cell smaller than the minimum grid point.

Examples

```
>>> g = TensorGrid([1, 2, 5], [-2, 1.5, 2], as_midp=False)
>>> g.min()
array([ 1., -2.])
>>> g = TensorGrid([1, 2, 5], [-2, 1.5, 2], as_midp=True)
>>> g.min()
array([ 0.5 , -3.75])
```

TensorGrid.points

`TensorGrid.points(`*order=None*`)`

All grid points in a single array.

Parametersorder : {'C', 'F'}

Force this axis ordering instead of the grid's own

Returnspoints : `numpy.ndarray`

The size of the array is ntotal x ndim, i.e. the points are stored as rows.

Examples

```
>>> g = TensorGrid([0, 1], [-1, 0, 2])  # default 'C' ordering
>>> g.points()
array([[ 0., -1.],
       [ 0.,  0.],
       [ 0.,  2.],
       [ 1., -1.],
       [ 1.,  0.],
       [ 1.,  2.]])
>>> g.points(order='F')
array([[ 0., -1.],
       [ 1., -1.],
       [ 0.,  0.],
       [ 1.,  0.],
       [ 0.,  2.],
       [ 1.,  2.]])
```

TensorGrid.size

`TensorGrid.`**`size`**`()`

Return a vector containing the total sizes.

TensorGrid.squeeze

`TensorGrid.`**`squeeze`**`()`

Remove the degenerate dimensions.

Note that no changes are made in-place.

Returnssqueezed : *TensorGrid*

The squeezed grid

Examples

```
>>> g = TensorGrid([0, 1], [-1], [-1, 0, 2])
>>> g.squeeze()
TensorGrid([0.0, 1.0], [-1.0, 0.0, 2.0])
```

`__init__`(**coord_vectors*, ***kwargs*)

Initialize a TensorGrid instance.

Parametersv1,...,vn : array-like

The coordinate vectors defining the grid points. They must be sorted in ascending order and may not contain duplicates. Empty vectors are not allowed.

kwargs : {'as_midp', 'order'}

'as_midp'[`bool`, optional (Default: `False`)] Treat grid points as midpoints of rectangular cells. This influences the behavior of *min*, *max* and *cell_sizes*.

'order'[{'C', 'F'}, optional] Ordering of the grid axes. 'C' means the first axis varies slowest, the last axis fastest; vice versa for 'F'.

Examples

```
>>> g = TensorGrid([1, 2, 5], [-2, 1.5, 2])
>>> g
TensorGrid([1.0, 2.0, 5.0], [-2.0, 1.5, 2.0])
>>> print(g)
grid [1.0, 2.0, 5.0] x [-2.0, 1.5, 2.0]
>>> g.ndim  # number of axes
2
>>> g.shape  # points per axis
(3, 3)
>>> g.ntotal  # total number of points
9
```

Grid points can be extracted with index notation (NOTE: This is slow, do not loop over the grid using indices!):

```
>>> g = TensorGrid([-1, 0, 3], [2, 4], [5], [2, 4, 7])
>>> g[0, 0, 0, 0]
array([-1.,  2.,  5.,  2.])
```

Slices and ellipsis are also supported:

```
>>> g[:, 0, 0, 0]
TensorGrid([-1.0, 0.0, 3.0], [2.0], [5.0], [2.0])
>>> g[0, ..., 1:]
TensorGrid([-1.0], [2.0, 4.0], [5.0], [4.0, 7.0])
```

Functions

sparse_meshgrid(*x, **kwargs)	Make a sparse meshgrid with C- or F-contiguous arrays.
uniform_sampling(intv_prod, num_nodes[, as_midp])	Sample an interval product uniformly.

sparse_meshgrid

odl.discr.grid.**sparse_meshgrid**(**x*, ***kwargs*)

Make a sparse meshgrid with C- or F-contiguous arrays.

uniform_sampling

odl.discr.grid.**uniform_sampling**(*intv_prod*, *num_nodes*, *as_midp=True*)

Sample an interval product uniformly.

Parametersintv_prod : *IntervalProd*

Set to be sampled

num_nodes : `int` or `tuple` of `int`

Number of nodes per axis. For dimension >= 2, a `tuple` is required. All entries must be positive. Entries corresponding to degenerate axes must be equal to 1.

as_midp : `bool`, optional

If `True`, the midpoints of an interval partition will be returned, which excludes the endpoints. Otherwise, equispaced nodes including the endpoints are generated. Note that the resulting strides are different. Default: `True`.

Returnssampling : *RegularGrid*

Uniform sampling grid for the interval product

Examples

```
>>> from odl import IntervalProd
>>> rbox = IntervalProd([-1.5, 2], [-0.5, 3])
>>> grid = uniform_sampling(rbox, [2, 5])
>>> grid.coord_vectors
(array([-1.25, -0.75]), array([ 2.1,  2.3,  2.5,  2.7,  2.9]))
>>> grid = uniform_sampling(rbox, [2, 5], as_midp=False)
>>> grid.coord_vectors
(array([-1.5, -0.5]), array([ 2.  ,  2.25,  2.5 ,  2.75,  3.  ]))
```

lp_discr

L^p type discretizations of function spaces.

Classes

DiscreteLp(fspace, grid, dspace[, exponent, ...])	Discretization of a Lebesgue L^p space.
DiscreteLpVector(space, data)	Representation of a *DiscreteLp* element.

DiscreteLp

class `odl.discr.lp_discr.`**`DiscreteLp`**(*fspace*, *grid*, *dspace*, *exponent=2.0*, *interp='nearest'*, ***kwargs*)

Bases: *odl.discr.discretization.Discretization*

Discretization of a Lebesgue L^p space.

Attributes

cell_size	Cell size of an underlying regular grid.
cell_volume	Cell volume of an underlying regular grid.
domain	The domain of the continuous space.
dspace	The data space.
dspace_type	Data space type of this discretization.
dtype	The data type of the representation space.
element_type	*DiscreteLpVector*
exponent	The exponent p in L^p.
extension	The operator mapping an n-tuple to a *uspace* element.

Continued on next page

Table 2.44 – continued from previous page

field	The field of this vector space.
grid	Sampling grid of the discretization mappings.
interp	Interpolation type of this discretization.
is_cn	If the space represents the set C^n.
is_rn	If the space represents the set R^n.
ndim	Number of dimensions.
order	Axis ordering for array flattening.
real_dtype	The corresponding real data type of this space.
restriction	The operator mapping a *uspace* element to an n-tuple.
shape	Shape of the underlying grid.
size	The number of entries per tuple.
uspace	The undiscretized space.

DiscreteLp.cell_size
DiscreteLp.**cell_size**
Cell size of an underlying regular grid.

DiscreteLp.cell_volume
DiscreteLp.**cell_volume**
Cell volume of an underlying regular grid.

DiscreteLp.domain
DiscreteLp.**domain**
The domain of the continuous space.

DiscreteLp.dspace
DiscreteLp.**dspace**
The data space.

DiscreteLp.dspace_type
DiscreteLp.**dspace_type**
Data space type of this discretization.

DiscreteLp.dtype
DiscreteLp.**dtype**
The data type of the representation space.

DiscreteLp.element_type
DiscreteLp.**element_type**
DiscreteLpVector

DiscreteLp.exponent
DiscreteLp.**exponent**
The exponent p in L^p.

DiscreteLp.extension
DiscreteLp.**extension**
The operator mapping an n-tuple to a *uspace* element.

DiscreteLp.field
DiscreteLp.**field**
The field of this vector space.

The field is the set of scalars of the space, that is numbers that the vectors in the space can be multiplied with.

Returnsfield : *Field*

The underlying field.

DiscreteLp.grid
DiscreteLp.**grid**
Sampling grid of the discretization mappings.

DiscreteLp.interp
DiscreteLp.**interp**
Interpolation type of this discretization.

DiscreteLp.is_cn
DiscreteLp.**is_cn**
If the space represents the set C^n.

Tuples of complex numbers.

DiscreteLp.is_rn
DiscreteLp.**is_rn**
If the space represents the set R^n.

Tuples of real numbers.

DiscreteLp.ndim
DiscreteLp.**ndim**
Number of dimensions.

DiscreteLp.order
DiscreteLp.**order**
Axis ordering for array flattening.

DiscreteLp.real_dtype
DiscreteLp.**real_dtype**
The corresponding real data type of this space.

DiscreteLp.restriction
DiscreteLp.**restriction**
The operator mapping a *uspace* element to an n-tuple.

DiscreteLp.shape
`DiscreteLp.`**`shape`**
Shape of the underlying grid.

DiscreteLp.size
`DiscreteLp.`**`size`**
The number of entries per tuple.

DiscreteLp.uspace
`DiscreteLp.`**`uspace`**
The undiscretized space.

Methods

element([inp])	Create an element from `inp` or from scratch.
points()	All points in the sampling grid.

DiscreteLp.element
`DiscreteLp.`**`element`**(*inp=None*)
Create an element from `inp` or from scratch.

Parametersinp : `object`, optional

The input data to create an element from. Must be recognizable by the *LinearSpace.element* method of either *RawDiscretization.dspace* or *RawDiscretization.uspace*.

Returnselement : *DiscreteLpVector*

The discretized element, calculated as `dspace.element(inp)` or `restriction(uspace.element(inp))`, tried in this order.

DiscreteLp.points
`DiscreteLp.`**`points`**()
All points in the sampling grid.
`__init__`(*fspace*, *grid*, *dspace*, *exponent=2.0*, *interp='nearest'*, ***kwargs*)
Initialize a new instance.

Parametersfspace : *FunctionSpace*

The continuous space to be discretized

grid : *TensorGrid*

The sampling grid for the discretization. Must be contained in `fspace.domain`.

dspace : *FnBase*

Space of elements used for data storage. It must have the same *FnBase.field* as `fspace`

exponent : positive `float`, optional

The parameter p in L^p. If the exponent is not equal to the default 2.0, the space has no inner product.

interp : `str`, optional

The interpolation type to be used for discretization.

'nearest' : use nearest-neighbor interpolation (default)

'linear' : use linear interpolation (not implemented)

kwargs : {'order'}

'order'[{'C', 'F'}, optional (Default: 'C')] Ordering of the values in the flat data arrays. 'C' means the first grid axis varies fastest, the last most slowly, 'F' vice versa.

DiscreteLpVector

class `odl.discr.lp_discr.`**`DiscreteLpVector`**(*space*, *data*)

Bases: *odl.discr.discretization.DiscretizationVector*

Representation of a *DiscreteLp* element.

Attributes

T	The transpose of a vector, the functional given by (.
cell_size	Cell size of an underlying regular grid.
cell_volume	Cell volume of an underlying regular grid.
dtype	type of data storage.
itemsize	The size in bytes on one element of this type.
nbytes	The number of bytes this vector uses in memory.
ndim	Number of dimensions.
ntuple	Structure for data storage.
shape	Multi-dimensional shape of this discrete function.
size	size of data storage.
space	Space to which this vector.
ufunc	*DiscreteLpUFuncs*, access to numpy style ufuncs.

DiscreteLpVector.T

`DiscreteLpVector.`**`T`**

The transpose of a vector, the functional given by (. , self)

Returnstranspose : *InnerProductOperator*

Notes

This function is only defined in inner product spaces.

In a complex space, this takes the conjugate transpose of the vector.

Examples

```
>>> from odl import Rn
>>> import numpy as np
>>> rn = Rn(3)
>>> x = rn.element([1, 2, 3])
```

```
        >>> y = rn.element([2, 1, 3])
        >>> x.T(y)
        13.0
```

DiscreteLpVector.cell_size
`DiscreteLpVector.cell_size`
Cell size of an underlying regular grid.

DiscreteLpVector.cell_volume
`DiscreteLpVector.cell_volume`
Cell volume of an underlying regular grid.

DiscreteLpVector.dtype
`DiscreteLpVector.dtype`
type of data storage.

DiscreteLpVector.itemsize
`DiscreteLpVector.itemsize`
The size in bytes on one element of this type.

DiscreteLpVector.nbytes
`DiscreteLpVector.nbytes`
The number of bytes this vector uses in memory.

DiscreteLpVector.ndim
`DiscreteLpVector.ndim`
Number of dimensions.

DiscreteLpVector.ntuple
`DiscreteLpVector.ntuple`
Structure for data storage.

DiscreteLpVector.shape
`DiscreteLpVector.shape`
Multi-dimensional shape of this discrete function.

DiscreteLpVector.size
`DiscreteLpVector.size`
size of data storage.

DiscreteLpVector.space
`DiscreteLpVector.space`
Space to which this vector.

DiscreteLpVector.ufunc
`DiscreteLpVector.ufunc`
DiscreteLpUFuncs, access to numpy style ufuncs.

Notes

These are optimized to use the underlying ntuple space and incur no overhead unless these do.

Examples

```
>>> X = uniform_discr(0, 1, 2)
>>> x = X.element([1, -2])
>>> x.ufunc.absolute()
uniform_discr(0.0, 1.0, 2).element([1.0, 2.0])
```

These functions can also be used with broadcasting

```
>>> x.ufunc.add(3)
uniform_discr(0.0, 1.0, 2).element([4.0, 1.0])
```

and non-space elements

```
>>> x.ufunc.subtract([3, 3])
uniform_discr(0.0, 1.0, 2).element([-2.0, -5.0])
```

There is also support for various reductions (sum, prod, min, max)

```
>>> x.ufunc.sum()
-1.0
```

Also supports out parameter

```
>>> y = X.element([3, 4])
>>> out = X.element()
>>> result = x.ufunc.add(y, out=out)
>>> result
uniform_discr(0.0, 1.0, 2).element([4.0, 2.0])
>>> result is out
True
```

Methods

__setitem__(indices, values)	Set values of this vector.
asarray([out])	Extract the data of this array as a numpy array.
show([method, title, indices])	Create a figure displaying the function in 1d or 2d.

DiscreteLpVector.__setitem__

`DiscreteLpVector.__setitem__`(*indices*, *values*)

Set values of this vector.

Parametersindices : `int` or `slice`

The position(s) that should be set

values : {scalar, array-like, *NtuplesVector*}

The value(s) that are to be assigned. If `indices` is an `int`, `values` must be a single value. If `indices` is a `slice`, `values` must be broadcastable to the size of the slice (same size, shape `(1,)` or single value). For `indices==slice(None, None,`

None), i.e. in the call vec[:] = values, a multi-dimensional array of correct shape is allowed as values.

DiscreteLpVector.asarray

DiscreteLpVector.**asarray**(*out=None*)

Extract the data of this array as a numpy array.

Parametersout : numpy.ndarray, optional

Array in which the result should be written in-place. Has to be contiguous and of the correct dtype and shape.

DiscreteLpVector.show

DiscreteLpVector.**show**(*method=''*, *title=''*, *indices=None*, ***kwargs*)

Create a figure displaying the function in 1d or 2d.

Parametersmethod : str, optional

1d methods:

'plot' : graph plot

2d methods:

'imshow' : image plot with coloring according to value,

including a colorbar.

'scatter' : cloud of scattered 3d points

(3rd axis <-> value)

indices : index expression

Display a slice of the array instead of the full array. The index expression is most easily created with the numpy.s_ constructur, i.e. supply np.s_[:, 1, :] to display the first slice along the second axis.

title : str, optional

Set the title of the figure

kwargs : {'figsize', 'saveto', ...}

Extra keyword arguments passed on to display method See the Matplotlib functions for documentation of extra options.

title : str, optional

Set the title of the figure

kwargs : {'figsize', 'saveto', ...}

Extra keyword arguments passed on to display method See the Matplotlib functions for documentation of extra options.

See also:

matplotlib.pyplot.plot Show graph plot

matplotlib.pyplot.imshow Show data as image

matplotlib.pyplot.scatter Show scattered 3d points

__init__ (*space*, *data*)
Initialize a new instance.

Functions

uniform_discr(min_corner, max_corner, nsamples)	Discretize an Lp function space by uniform sampling.
uniform_discr_fromspace(fspace, nsamples[, ...])	Discretize an Lp function space by uniform sampling.

uniform_discr

odl.discr.lp_discr.**uniform_discr**(*min_corner*, *max_corner*, *nsamples*, *exponent=2.0*, *field=RealNumbers()*, *interp='nearest'*, *impl='numpy'*, ***kwargs*)

Discretize an Lp function space by uniform sampling.

Parametersmin_corner : `float` or `tuple` of `float`

Minimum corner of the result.

nsamples : `float` or `tuple` of `float`

Minimum corner of the result.

nsamples : `int` or `tuple` of `int`

Number of samples per axis. For dimension >= 2, a tuple is required.

exponent : positive `float`, optional

The parameter p in L^p. If the exponent is not equal to the default 2.0, the space has no inner product.

field : *Field*, optional

The field of the *FunctionSpace*, default *RealNumbers*.

interp : `str`, optional

Interpolation type to be used for discretization.

'nearest' : use nearest-neighbor interpolation (default)

'linear' : use linear interpolation (not implemented)

impl : {'numpy', 'cuda'}

Implementation of the data storage arrays

kwargs : {'order', 'dtype', 'weighting'}

'order'[{'C', 'F'} (Default: 'C')] Axis ordering in the data storage

'dtype'[dtype] Data type for the discretized space

Default for 'numpy': 'float64' / 'complex128' Default for 'cuda': 'float32' / TODO

'weighting'[{'simple', 'consistent'}] Weighting of the discretized space functions.

'simple': weight is a constant (cell volume)

'consistent': weight is a matrix depending on the interpolation type

Returnsdiscr : *DiscreteLp*

The uniformly discretized function space

See also:

uniform_discr_fromspace

Examples

Create real space:

```
>>> uniform_discr([0, 0], [1, 1], [10, 10])
uniform_discr([0.0, 0.0], [1.0, 1.0], [10, 10])
```

```
>>> from odl import ComplexNumbers
>>> uniform_discr([0, 0], [1, 1], [10, 10], field=ComplexNumbers())
uniform_discr([0.0, 0.0], [1.0, 1.0], [10, 10], field=ComplexNumbers())
```

uniform_discr_fromspace

odl.discr.lp_discr.**uniform_discr_fromspace**(*fspace*, *nsamples*, *exponent=2.0*, *interp='nearest'*, *impl='numpy'*, ***kwargs*)

Discretize an Lp function space by uniform sampling.

Parametersfspace : *FunctionSpace*

Continuous function space. Its domain must be an *IntervalProd* instance.

nsamples : `int` or `tuple` of `int`

Number of samples per axis. For dimension >= 2, a tuple is required.

exponent : positive `float`, optional

The parameter p in L^p. If the exponent is not equal to the default 2.0, the space has no inner product.

interp : `str`, optional

Interpolation type to be used for discretization.

'nearest' : use nearest-neighbor interpolation (default)

'linear' : use linear interpolation (not implemented)

impl : {'numpy', 'cuda'}

Implementation of the data storage arrays

kwargs : {'order', 'dtype', 'weighting'}

'order'[{'C', 'F'} (Default: 'C')] Axis ordering in the data storage

'dtype'[dtype] Data type for the discretized space

Default for 'numpy': 'float64' / 'complex128' Default for 'cuda': 'float32' / TODO

'weighting'[{'simple', 'consistent'}] Weighting of the discretized space functions.

'simple': weight is a constant (cell volume)

'consistent': weight is a matrix depending on the interpolation type

Returnsdiscr : *DiscreteLp*

The uniformly discretized function space

See also:

uniform_discr

Examples

```
>>> from odl import Interval, FunctionSpace
>>> I = Interval(0, 1)
>>> X = FunctionSpace(I)
>>> uniform_discr_fromspace(X, 10)
uniform_discr(0.0, 1.0, 10)
```

2.4.3 operator

Mathematical operators in ODL.

Modules

default_ops

Default operators defined on any (reasonable) space.

Classes

ConstantOperator(vector[, dom])	Operator that always returns the same value
IdentityOperator(space)	Operator mapping each element to itself.
InnerProductAdjointOperator(vector)	Operator taking the scalar product with a fixed vector.
InnerProductOperator(vector)	Operator taking the inner product with a fixed vector.
LinCombOperator(space, a, b)	Operator mapping two space elements to a linear combination.
MultiplyOperator(space)	Operator multiplying two elements.
ResidualOperator(op, vector)	Operator that returns the residual of some operator application
ScalingOperator(space, scalar)	Operator of multiplication with a scalar.
ZeroOperator(space)	Operator mapping each element to the zero element.

ConstantOperator

class odl.operator.default_ops.**ConstantOperator**(*vector*, *dom=None*)

Bases: *odl.operator.operator.Operator*

Operator that always returns the same value

```
ConstantOperator(vector)(x) <==> vector
```

Attributes

adjoint	The operator adjoint (abstract).

Continued on next page

Table 2.50 – continued from previous page

domain	The domain of this operator.
inverse	Return the operator inverse.
is_functional	`True` if the this operator's range is a *Field*.
is_linear	`True` if this operator is linear.
range	The range of this operator.

ConstantOperator.adjoint

`ConstantOperator.`**`adjoint`**

The operator adjoint (abstract).

RaisesOpNotImplementedError

Since the adjoint cannot be default implemented.

ConstantOperator.domain

`ConstantOperator.`**`domain`**

The domain of this operator.

ConstantOperator.inverse

`ConstantOperator.`**`inverse`**

Return the operator inverse.

RaisesOpNotImplementedError

Since the inverse cannot be default implemented.

ConstantOperator.is_functional

`ConstantOperator.`**`is_functional`**

`True` if the this operator's range is a *Field*.

ConstantOperator.is_linear

`ConstantOperator.`**`is_linear`**

`True` if this operator is linear.

ConstantOperator.range

`ConstantOperator.`**`range`**

The range of this operator.

Methods

`__eq__`	Return self==value.
_apply(x, out)	Assign out to the constant vector
_call(x)	Returns the constant vector

ConstantOperator._apply

`ConstantOperator.`**`_apply`**(*x*, *out*)

Assign out to the constant vector

Parametersx : `domain` element

Any element in the domain

out : `range` element

Vector that gets assigned to the constant vector

Examples

```
>>> from odl import Rn
>>> r3 = Rn(3)
>>> x = r3.element([1, 2, 3])
>>> op = ConstantOperator(x)
>>> op(x, out=r3.element())
Rn(3).element([1.0, 2.0, 3.0])
```

ConstantOperator._call

`ConstantOperator.`**`_call`**`(`*x*`)`

Returns the constant vector

Parametersx : `domain` element

Any element in the domain

Returnsvector : *LinearSpaceVector*

The constant vector

Examples

```
>>> from odl import Rn
>>> r3 = Rn(3)
>>> x = r3.element([1, 2, 3])
>>> op = ConstantOperator(x)
>>> op(x)
Rn(3).element([1.0, 2.0, 3.0])
>>> op(r3.zero())
Rn(3).element([1.0, 2.0, 3.0])
>>> op(r3.element())
Rn(3).element([1.0, 2.0, 3.0])
```

`__init__``(`*vector*, *dom=None*`)`

Initialize an instance.

Parametersvector : *LinearSpaceVector*

The vector constant to be returned

dom : *LinearSpace*, default

The domain of the operator.

IdentityOperator

class `odl.operator.default_ops.`**`IdentityOperator`**`(`*space*`)`

Bases: *odl.operator.default_ops.ScalingOperator*

Operator mapping each element to itself.

Attributes

adjoint	The adjoint is given by taking the conjugate of the scalar
domain	The domain of this operator.
inverse	Return the inverse operator.
is_functional	`True` if the this operator's range is a *Field*.
is_linear	`True` if this operator is linear.
range	The range of this operator.

IdentityOperator.adjoint

`IdentityOperator.`**`adjoint`**

The adjoint is given by taking the conjugate of the scalar

IdentityOperator.domain

`IdentityOperator.`**`domain`**

The domain of this operator.

IdentityOperator.inverse

`IdentityOperator.`**`inverse`**

Return the inverse operator.

Examples

```
>>> from odl import Rn
>>> r3 = Rn(3)
>>> vec = r3.element([1, 2, 3])
>>> op = ScalingOperator(r3, 2.0)
>>> inv = op.inverse
>>> inv(op(vec)) == vec
True
>>> op(inv(vec)) == vec
True
```

IdentityOperator.is_functional

`IdentityOperator.`**`is_functional`**

`True` if the this operator's range is a *Field*.

IdentityOperator.is_linear

`IdentityOperator.`**`is_linear`**

`True` if this operator is linear.

IdentityOperator.range

`IdentityOperator.`**`range`**

The range of this operator.

Methods

__eq__	Return self==value.
_apply(x, out)	Scale input and write to output.
_call(x)	Return the scaled element.

IdentityOperator._apply

IdentityOperator.**_apply**(*x*, *out*)

Scale input and write to output.

Parametersx : domain element

input vector to be scaled

out : range element

Output vector to which the result is written

ReturnsNone

Examples

```
>>> from odl import Rn
>>> r3 = Rn(3)
>>> vec = r3.element([1, 2, 3])
>>> out = r3.element()
>>> op = ScalingOperator(r3, 2.0)
>>> op(vec, out)  # Returns out
Rn(3).element([2.0, 4.0, 6.0])
>>> out
Rn(3).element([2.0, 4.0, 6.0])
```

IdentityOperator._call

IdentityOperator.**_call**(*x*)

Return the scaled element.

Parametersx : domain element

input vector to be scaled

Returnsscaled : range element

The scaled vector

Examples

```
>>> from odl import Rn
>>> r3 = Rn(3)
>>> vec = r3.element([1, 2, 3])
>>> op = ScalingOperator(r3, 2.0)
>>> op(vec)
Rn(3).element([2.0, 4.0, 6.0])
```

__init__(*space*)

Initialize an IdentityOperator instance.

Parametersspace : LinearSpace

The space of elements which the operator is acting on

InnerProductAdjointOperator

class odl.operator.default_ops.**InnerProductAdjointOperator**(*vector*)
Bases: *odl.operator.operator.Operator*

Operator taking the scalar product with a fixed vector.

The multiply operator calculates:

```
InnerProductAdjointOperator(vec)(x) == x * vec
```

Attributes

adjoint	
domain	The domain of this operator.
inverse	Return the operator inverse.
is_functional	`True` if the this operator's range is a *Field*.
is_linear	`True` if this operator is linear.
range	The range of this operator.

InnerProductAdjointOperator.adjoint
InnerProductAdjointOperator.**adjoint**

InnerProductAdjointOperator.domain
InnerProductAdjointOperator.**domain**
The domain of this operator.

InnerProductAdjointOperator.inverse
InnerProductAdjointOperator.**inverse**
Return the operator inverse.

RaisesOpNotImplementedError

Since the inverse cannot be default implemented.

InnerProductAdjointOperator.is_functional
InnerProductAdjointOperator.**is_functional**
`True` if the this operator's range is a *Field*.

InnerProductAdjointOperator.is_linear
InnerProductAdjointOperator.**is_linear**
`True` if this operator is linear.

InnerProductAdjointOperator.range
InnerProductAdjointOperator.**range**
The range of this operator.

Methods

__eq__	Return self==value.
_apply(x, out)	Multiply the input and write to output.
_call(x)	Multiply by the input.

InnerProductAdjointOperator._apply

InnerProductAdjointOperator.**_apply**(*x*, *out*)

Multiply the input and write to output.

Parametersx : domain element

An element in the operator domain (2-tuple of space elements) whose elementwise product is calculated

out : range element

Vector to which the result is written

Examples

```
>>> from odl import Rn
>>> r3 = Rn(3)
>>> x = r3.element([1, 2, 3])
>>> op = InnerProductAdjointOperator(x)
>>> out = r3.element()
>>> result = op(3.0, out=out)
>>> result
Rn(3).element([3.0,  6.0,  9.0])
>>> result is out
True
```

InnerProductAdjointOperator._call

InnerProductAdjointOperator.**_call**(*x*)

Multiply by the input.

Parametersx : vector.space.field element

An element in the field of the vector

Examples

```
>>> from odl import Rn
>>> r3 = Rn(3)
>>> x = r3.element([1, 2, 3])
>>> op = InnerProductAdjointOperator(x)
>>> op(3.0)
Rn(3).element([3.0, 6.0, 9.0])
```

__init__(*vector*)

Initialize a InnerProductOperator instance.

Parametersvector : *LinearSpaceVector*

The vector to take the inner product with

InnerProductOperator

class odl.operator.default_ops.**InnerProductOperator**(*vector*)

Bases: *odl.operator.operator.Operator*

Operator taking the inner product with a fixed vector.

The multiply operator calculates:

```
InnerProductOperator(vec)(x) == x.inner(vec)
```

This is only applicable in inner product spaces.

Attributes

T	
adjoint	
domain	The domain of this operator.
inverse	Return the operator inverse.
is_functional	`True` if the this operator's range is a *Field*.
is_linear	`True` if this operator is linear.
range	The range of this operator.

InnerProductOperator.T
InnerProductOperator.**T**

InnerProductOperator.adjoint
InnerProductOperator.**adjoint**

InnerProductOperator.domain
InnerProductOperator.**domain**
The domain of this operator.

InnerProductOperator.inverse
InnerProductOperator.**inverse**
Return the operator inverse.

RaisesOpNotImplementedError

Since the inverse cannot be default implemented.

InnerProductOperator.is_functional
InnerProductOperator.**is_functional**
`True` if the this operator's range is a *Field*.

InnerProductOperator.is_linear
InnerProductOperator.**is_linear**
`True` if this operator is linear.

InnerProductOperator.range
`InnerProductOperator.`**`range`**
The range of this operator.

Methods

`__eq__`	Return self==value.
`_call`(x)	Multiply the input and write to output.

InnerProductOperator._call
`InnerProductOperator.`**`_call`**(*x*)
Multiply the input and write to output.

Parametersx : `vector.space` element

An element in the space of the vector

Examples

```
>>> from odl import Rn
>>> r3 = Rn(3)
>>> x = r3.element([1, 2, 3])
>>> op = InnerProductOperator(x)
>>> op(r3.element([1, 2, 3]))
14.0
```

`__init__`(*vector*)
Initialize a InnerProductOperator instance.

Parametersvector : *`LinearSpaceVector`*

The vector to take the inner product with

LinCombOperator

class `odl.operator.default_ops.`**`LinCombOperator`**(*space*, *a*, *b*)
Bases: *`odl.operator.operator.Operator`*

Operator mapping two space elements to a linear combination.

This opertor calculates:

```
out = a*x[0] + b*x[1]
```

Attributes

`adjoint`	The operator adjoint (abstract).
`domain`	The domain of this operator.
`inverse`	Return the operator inverse.
`is_functional`	`True` if the this operator's range is a *`Field`*.
`is_linear`	`True` if this operator is linear.
`range`	The range of this operator.

LinCombOperator.adjoint
`LinCombOperator.`**`adjoint`**
The operator adjoint (abstract).

RaisesOpNotImplementedError

Since the adjoint cannot be default implemented.

LinCombOperator.domain
`LinCombOperator.`**`domain`**
The domain of this operator.

LinCombOperator.inverse
`LinCombOperator.`**`inverse`**
Return the operator inverse.

RaisesOpNotImplementedError

Since the inverse cannot be default implemented.

LinCombOperator.is_functional
`LinCombOperator.`**`is_functional`**
`True` if the this operator's range is a *`Field`*.

LinCombOperator.is_linear
`LinCombOperator.`**`is_linear`**
`True` if this operator is linear.

LinCombOperator.range
`LinCombOperator.`**`range`**
The range of this operator.

Methods

`__eq__`	Return self==value.
`_apply`(x, out)	Linearly combine the input and write to output.

LinCombOperator._apply
`LinCombOperator.`**`_apply`**(*x*, *out*)
Linearly combine the input and write to output.

Parametersx : `domain` element

An element in the operator domain (2-tuple of space elements) whose linear combination is calculated

out : `'range` element

Vector to which the result is written

Examples

```
>>> from odl import Rn, ProductSpace
>>> r3 = Rn(3)
>>> r3xr3 = ProductSpace(r3, r3)
>>> xy = r3xr3.element([[1, 2, 3], [1, 2, 3]])
>>> z = r3.element()
>>> op = LinCombOperator(r3, 1.0, 1.0)
>>> op(xy, out=z)  # Returns z
Rn(3).element([2.0, 4.0, 6.0])
>>> z
Rn(3).element([2.0, 4.0, 6.0])
```

__init__(*space*, *a*, *b*)

Initialize a LinCombOperator instance.

Parametersspace : *LinearSpace*

The space of elements which the operator is acting on

a : scalar

Scalar to multiply x[0] with

b : scalar

Scalar to multiply x[1] with

MultiplyOperator

class odl.operator.default_ops.**MultiplyOperator**(*space*)

Bases: *odl.operator.operator.Operator*

Operator multiplying two elements.

The multiply operator calculates:

out = x[0] * x[1]

This is only applicable in Algebras.

Attributes

adjoint	The operator adjoint (abstract).
domain	The domain of this operator.
inverse	Return the operator inverse.
is_functional	`True` if the this operator's range is a *Field*.
is_linear	`True` if this operator is linear.
range	The range of this operator.

MultiplyOperator.adjoint

MultiplyOperator.**adjoint**

The operator adjoint (abstract).

RaisesOpNotImplementedError

Since the adjoint cannot be default implemented.

MultiplyOperator.domain
`MultiplyOperator.`**`domain`**
The domain of this operator.

MultiplyOperator.inverse
`MultiplyOperator.`**`inverse`**
Return the operator inverse.

RaisesOpNotImplementedError

Since the inverse cannot be default implemented.

MultiplyOperator.is_functional
`MultiplyOperator.`**`is_functional`**
`True` if the this operator's range is a *`Field`*.

MultiplyOperator.is_linear
`MultiplyOperator.`**`is_linear`**
`True` if this operator is linear.

MultiplyOperator.range
`MultiplyOperator.`**`range`**
The range of this operator.

Methods

`__eq__`	Return self==value.
`_apply`(x, out)	Multiply the input and write to output.

MultiplyOperator._apply
`MultiplyOperator.`**`_apply`**(*x*, *out*)
Multiply the input and write to output.

Parametersx : `domain` element

An element in the operator domain (2-tuple of space elements) whose elementwise product is calculated

out : `range` element

Vector to which the result is written

Examples

```
>>> from odl import Rn
>>> r3 = Rn(3)
>>> r3xr3 = ProductSpace(r3, r3)
>>> xy = r3xr3.element([[1, 2, 3], [1, 2, 3]])
>>> z = r3.element()
>>> op = MultiplyOperator(r3)
>>> op(xy, z)
```

```
        Rn(3).element([1.0, 4.0, 9.0])
        >>> z
        Rn(3).element([1.0, 4.0, 9.0])
```

__init__(*space*)
Initialize a MultiplyOperator instance.

Parametersspace : LinearSpace

The space of elements which the operator is acting on

ResidualOperator

class odl.operator.default_ops.**ResidualOperator**(*op*, *vector*)
Bases: *odl.operator.operator.Operator*

Operator that returns the residual of some operator application with a vector

```
ResidualOperator(op, vector)(x) <==> op(x) - vector
```

Attributes

adjoint	The operator adjoint (abstract).
domain	The domain of this operator.
inverse	Return the operator inverse.
is_functional	`True` if the this operator's range is a *Field*.
is_linear	`True` if this operator is linear.
range	The range of this operator.

ResidualOperator.adjoint
ResidualOperator.**adjoint**
The operator adjoint (abstract).

RaisesOpNotImplementedError

Since the adjoint cannot be default implemented.

ResidualOperator.domain
ResidualOperator.**domain**
The domain of this operator.

ResidualOperator.inverse
ResidualOperator.**inverse**
Return the operator inverse.

RaisesOpNotImplementedError

Since the inverse cannot be default implemented.

ResidualOperator.is_functional
ResidualOperator.**is_functional**
`True` if the this operator's range is a *Field*.

ResidualOperator.is_linear
`ResidualOperator.`**`is_linear`**
`True` if this operator is linear.

ResidualOperator.range
`ResidualOperator.`**`range`**
The range of this operator.

Methods

`__eq__`	Return self==value.
`_apply`(x, out)	Assign out to the constant vector
`_call`(x)	Returns the constant vector
`derivative`(point)	The derivative of a residual is the derivative of the operator

ResidualOperator._apply
`ResidualOperator.`**`_apply`**(*x*, *out*)
Assign out to the constant vector

Parametersx : `domain` element

Any element in the domain

out : `range` element

Vector that gets assigned to the constant vector

Examples

```
>>> from odl import Rn
>>> r3 = Rn(3)
>>> vec = r3.element([1, 2, 3])
>>> op = IdentityOperator(r3)
>>> res = ResidualOperator(op, vec)
>>> x = r3.element([4, 5, 6])
>>> res(x, out=r3.element())
Rn(3).element([3.0, 3.0, 3.0])
```

ResidualOperator._call
`ResidualOperator.`**`_call`**(*x*)
Returns the constant vector

Parametersx : `domain` element

Any element in the domain

Returnsvector : *`LinearSpaceVector`*

The constant vector

Examples

```
>>> from odl import Rn
>>> r3 = Rn(3)
>>> vec = r3.element([1, 2, 3])
>>> op = IdentityOperator(r3)
>>> res = ResidualOperator(op, vec)
>>> x = r3.element([4, 5, 6])
>>> res(x)
Rn(3).element([3.0, 3.0, 3.0])
```

ResidualOperator.derivative

`ResidualOperator.`**`derivative`**(*point*)

The derivative of a residual is the derivative of the operator

```
ResidualOperator(op, vec).derivative(x) <==> op.derivative(x)
```

Parametersx : `domain` element

Any element in the domain where the derivative should be taken

`__init__`(*op*, *vector*)

Initialize an instance.

Parametersvector : *LinearSpaceVector*

The vector constant to be returned

dom : *LinearSpace*, default

The domain of the operator.

ScalingOperator

class `odl.operator.default_ops.`**`ScalingOperator`**(*space*, *scalar*)

Bases: *odl.operator.operator.Operator*

Operator of multiplication with a scalar.

Attributes

adjoint	The adjoint is given by taking the conjugate of the scalar
domain	The domain of this operator.
inverse	Return the inverse operator.
is_functional	`True` if the this operator's range is a *Field*.
is_linear	`True` if this operator is linear.
range	The range of this operator.

ScalingOperator.adjoint

`ScalingOperator.`**`adjoint`**

The adjoint is given by taking the conjugate of the scalar

ScalingOperator.domain

ScalingOperator.**domain**
The domain of this operator.

ScalingOperator.inverse
ScalingOperator.**inverse**
Return the inverse operator.

Examples

```
>>> from odl import Rn
>>> r3 = Rn(3)
>>> vec = r3.element([1, 2, 3])
>>> op = ScalingOperator(r3, 2.0)
>>> inv = op.inverse
>>> inv(op(vec)) == vec
True
>>> op(inv(vec)) == vec
True
```

ScalingOperator.is_functional
ScalingOperator.**is_functional**
`True` if the this operator's range is a *Field*.

ScalingOperator.is_linear
ScalingOperator.**is_linear**
`True` if this operator is linear.

ScalingOperator.range
ScalingOperator.**range**
The range of this operator.

Methods

__eq__	Return self==value.
_apply(x, out)	Scale input and write to output.
_call(x)	Return the scaled element.

ScalingOperator._apply
ScalingOperator.**_apply**(*x*, *out*)
Scale input and write to output.

Parameters **x** : domain element

input vector to be scaled

out : range element

Output vector to which the result is written

Returns None

Examples

```
>>> from odl import Rn
>>> r3 = Rn(3)
>>> vec = r3.element([1, 2, 3])
>>> out = r3.element()
>>> op = ScalingOperator(r3, 2.0)
>>> op(vec, out)  # Returns out
Rn(3).element([2.0, 4.0, 6.0])
>>> out
Rn(3).element([2.0, 4.0, 6.0])
```

ScalingOperator._call

`ScalingOperator.__call__`(*x*)

Return the scaled element.

Parametersx : domain element

input vector to be scaled

Returnsscaled : `range` element

The scaled vector

Examples

```
>>> from odl import Rn
>>> r3 = Rn(3)
>>> vec = r3.element([1, 2, 3])
>>> op = ScalingOperator(r3, 2.0)
>>> op(vec)
Rn(3).element([2.0, 4.0, 6.0])
```

__init__(*space*, *scalar*)

Initialize a ScalingOperator instance.

Parametersspace : *LinearSpace*

The space of elements which the operator is acting on

scalar : field element

An element in the field of the space that the vectors are scaled with

ZeroOperator

class odl.operator.default_ops.**ZeroOperator**(*space*)

Bases: *odl.operator.default_ops.ScalingOperator*

Operator mapping each element to the zero element.

Attributes

adjoint	The adjoint is given by taking the conjugate of the scalar

Continued on next page

Table 2.66 – continued from previous page

domain	The domain of this operator.
inverse	Return the inverse operator.
is_functional	`True` if the this operator's range is a *Field*.
is_linear	`True` if this operator is linear.
range	The range of this operator.

ZeroOperator.adjoint

`ZeroOperator.`**`adjoint`**

The adjoint is given by taking the conjugate of the scalar

ZeroOperator.domain

`ZeroOperator.`**`domain`**

The domain of this operator.

ZeroOperator.inverse

`ZeroOperator.`**`inverse`**

Return the inverse operator.

Examples

```
>>> from odl import Rn
>>> r3 = Rn(3)
>>> vec = r3.element([1, 2, 3])
>>> op = ScalingOperator(r3, 2.0)
>>> inv = op.inverse
>>> inv(op(vec)) == vec
True
>>> op(inv(vec)) == vec
True
```

ZeroOperator.is_functional

`ZeroOperator.`**`is_functional`**

`True` if the this operator's range is a *Field*.

ZeroOperator.is_linear

`ZeroOperator.`**`is_linear`**

`True` if this operator is linear.

ZeroOperator.range

`ZeroOperator.`**`range`**

The range of this operator.

Methods

`__eq__`	Return self==value.

Continued on next page

Table 2.67 – continued from previous page

_apply(x, out)	Scale input and write to output.
_call(x)	Return the scaled element.

ZeroOperator._apply

`ZeroOperator.`**`_apply`**(*x*, *out*)

Scale input and write to output.

Parametersx : `domain` element

input vector to be scaled

out : `range` element

Output vector to which the result is written

Returns`None`

Examples

```
>>> from odl import Rn
>>> r3 = Rn(3)
>>> vec = r3.element([1, 2, 3])
>>> out = r3.element()
>>> op = ScalingOperator(r3, 2.0)
>>> op(vec, out)  # Returns out
Rn(3).element([2.0, 4.0, 6.0])
>>> out
Rn(3).element([2.0, 4.0, 6.0])
```

ZeroOperator._call

`ZeroOperator.`**`_call`**(*x*)

Return the scaled element.

Parametersx : domain element

input vector to be scaled

Returnsscaled : `range` element

The scaled vector

Examples

```
>>> from odl import Rn
>>> r3 = Rn(3)
>>> vec = r3.element([1, 2, 3])
>>> op = ScalingOperator(r3, 2.0)
>>> op(vec)
Rn(3).element([2.0, 4.0, 6.0])
```

`__init__`(*space*)

Initialize a ZeroOperator instance.

Parametersspace : *LinearSpace*

The space of elements which the operator is acting on

operator

Abstract mathematical operators.

Classes

FunctionalLeftVectorMult(op, vector)	Expression type for the functional left vector multiplication.
OpDomainError	Exception for domain errors.
OpNotImplementedError	Exception for not implemented errors in *LinearSpace*'s.
OpRangeError	Exception for domain errors.
OpTypeError	Exception for operator type errors.
Operator(domain, range[, linear])	Abstract operator.
OperatorComp(left, right[, tmp])	Expression type for the composition of operators.
OperatorLeftScalarMult(op, scalar)	Expression type for the operator left scalar multiplication.
OperatorLeftVectorMult(op, vector)	Expression type for the operator left vector multiplication.
OperatorPointwiseProduct(op1, op2)	Expression type for the pointwise operator mulitplication.
OperatorRightScalarMult(op, scalar[, tmp])	Expression type for the operator right scalar multiplication.
OperatorRightVectorMult(op, vector)	Expression type for the operator right vector multiplication.
OperatorSum(op1, op2[, tmp_ran, tmp_dom])	Expression type for the sum of operators.

FunctionalLeftVectorMult

class odl.operator.operator.**FunctionalLeftVectorMult**(*op*, *vector*)

Bases: *odl.operator.operator.Operator*

Expression type for the functional left vector multiplication.

A functional is a *Operator* whose *Operator.range* is a *Field*.

```
FunctionalLeftVectorMult(op, vector)(x) <==> vector * op(x)
```

Attributes

adjoint	The operator adjoint.
domain	The domain of this operator.
inverse	Return the operator inverse.
is_functional	`True` if the this operator's range is a *Field*.
is_linear	`True` if this operator is linear.
range	The range of this operator.

FunctionalLeftVectorMult.adjoint

FunctionalLeftVectorMult.**adjoint**

The operator adjoint.

The adjoint of the operator scalar multiplication is the scalar multiplication of the operator adjoint:

```
FunctionalLeftVectorMult(op, vector).adjoint == OperatorComp(op.adjoint,
vector.T)
```

```
(x * A)^T = A^T * x^T
```

RaisesOpNotImplementedError

If the underlying operator is non-linear.

FunctionalLeftVectorMult.domain
`FunctionalLeftVectorMult.`**`domain`**
The domain of this operator.

FunctionalLeftVectorMult.inverse
`FunctionalLeftVectorMult.`**`inverse`**
Return the operator inverse.

RaisesOpNotImplementedError

Since the inverse cannot be default implemented.

FunctionalLeftVectorMult.is_functional
`FunctionalLeftVectorMult.`**`is_functional`**
`True` if the this operator's range is a *`Field`*.

FunctionalLeftVectorMult.is_linear
`FunctionalLeftVectorMult.`**`is_linear`**
`True` if this operator is linear.

FunctionalLeftVectorMult.range
`FunctionalLeftVectorMult.`**`range`**
The range of this operator.

Methods

`__call__`(x[, out])	Return `op(x)`.
`__eq__`	Return self==value.
`_apply`(x, out)	Implement `op(x, out=out)`.
`_call`(x)	Return `op(x)`.
`derivative`(x)	Return the derivative at `x`.

FunctionalLeftVectorMult.__call__
`FunctionalLeftVectorMult.`**`__call__`**(*x*, *out=None*, **args*, ***kwargs*)
Return `op(x)`.

Implementation of the call pattern `op(x)` with the private `_call()` method and added error checking.

Parametersx : *`Operator.domain`* element

An object in the operator domain to which the operator is applied. The object is treated as immutable, hence it is not modified during evaluation.

out : *`Operator.range`* element, optional

An object in the operator range to which the result of the operator evaluation is written. The result is independent of the initial state of this object.

***args, **kwargs** : Further arguments to the function, optional

Returnselem : *`Operator.range`* element

An object in the operator range, the result of the operator evaluation. It is identical to `out` if provided.

Examples

```
>>> from odl import Rn, ScalingOperator
>>> rn = Rn(3)
>>> op = ScalingOperator(rn, 2.0)
>>> x = rn.element([1, 2, 3])
```

Out-of-place evaluation:

```
>>> op(x)
Rn(3).element([2.0, 4.0, 6.0])
```

In-place evaluation:

```
>>> y = rn.element()
>>> op(x, out=y)
Rn(3).element([2.0, 4.0, 6.0])
>>> y
Rn(3).element([2.0, 4.0, 6.0])
```

FunctionalLeftVectorMult._apply

`FunctionalLeftVectorMult.`**`_apply`**(*x*, *out*)

Implement `op(x, out=out)`.

FunctionalLeftVectorMult._call

`FunctionalLeftVectorMult.`**`_call`**(*x*)

Return `op(x)`.

FunctionalLeftVectorMult.derivative

`FunctionalLeftVectorMult.`**`derivative`**(*x*)

Return the derivative at `x`.

Left scalar multiplication and derivative are commutative:

```
FunctionalLeftVectorMult(op, vector).derivative(x) <==>
FunctionalLeftVectorMult(op.derivative(x), vector)
```

See also:

FunctionalLeftVectorMultthe result

__init__(*op*, *vector*)

Initialize a new *FunctionalLeftVectorMult* instance.

Parametersop : *Operator*

The range of `op` must be a *Field*.

vector : *LinearSpaceVector*

The vector to multiply by. its space's *LinearSpace.field* must be the same as `op.range`

OpDomainError

exception odl.operator.operator.**OpDomainError**
Exception for domain errors.

Domain errors are raised by *Operator* subclasses when trying to call them with input not in the domain (*Operator.domain*).

OpNotImplementedError

exception odl.operator.operator.**OpNotImplementedError**
Exception for not implemented errors in *LinearSpace*'s.

These are raised when a method in *LinearSpace* that has not been defined in a specific space is called.

OpRangeError

exception odl.operator.operator.**OpRangeError**
Exception for domain errors.

Domain errors are raised by *Operator* subclasses when the returned value does not lie in the range (*Operator.range*).

OpTypeError

exception odl.operator.operator.**OpTypeError**
Exception for operator type errors.

Domain errors are raised by *Operator* subclasses when trying to call them with input not in the domain (*Operator.domain*) or with the wrong range (*Operator.range*).

Operator

class odl.operator.operator.**Operator**(*domain*, *range*, *linear=False*)
Bases: object

Abstract operator.

Abstract attributes and methods

Operator is an **abstract** class, i.e. it can only be subclassed, not used directly.

Any subclass of *Operator* **must** have the following attributes:

domain[*Set*] The set of elements this operator can be applied to

range[*Set*] The set this operator maps to

It is **highly** recommended to call `super().__init__(dom, ran)` (Note: add `from builtins import super` in Python 2) in the `__init__()` method of any subclass, where `dom` and `ran` are the arguments specifying domain and range of the new operator. In that case, the attributes *Operator.domain* and *Operator.range* are automatically provided by *Operator*.

In addition, any subclass **must** implement **at least one** of the methods `_apply()` and `_call()`, which are explained in the following.

In-place evaluation: `_apply()`

In-place evaluation means that the operator is applied, and the result is written to an existing element provided as an additional argument. In this case, a subclass has to implement the method

```
_apply(self, x, out) <==> out <-- operator(x)
```

Parameters:

x[*Operator.domain* element] An object in the operator domain to which the operator is applied.

out[*Operator.range* element] An object in the operator range to which the result of the operator evaluation is written.

Returns:

`None`

Out-of-place evaluation: `_call()`

Out-of-place evaluation means that the operator is applied, and the result is written to a **new** element which is returned. In this case, a subclass has to implement the method

```
_call(self, x) <==> operator(x)
```

Parameters:

x[*Operator.domain* element] An object in the operator domain to which the operator is applied.

out[*Operator.range* element] An object in the operator range to which the result of the operator evaluation is written.

Notes

If not both `_apply()` and `_call()` are implemented and the *Operator.range* is a *LinearSpace*, a default implementation of the respective other is provided.

Attributes

adjoint	The operator adjoint (abstract).
domain	The domain of this operator.
inverse	Return the operator inverse.
is_functional	`True` if the this operator's range is a *Field*.
is_linear	`True` if this operator is linear.
range	The range of this operator.

Operator.adjoint

`Operator.`**`adjoint`**

The operator adjoint (abstract).

RaisesOpNotImplementedError

Since the adjoint cannot be default implemented.

Operator.domain

`Operator.`**`domain`**

The domain of this operator.

Operator.inverse
`Operator.`**`inverse`**
Return the operator inverse.

RaisesOpNotImplementedError

Since the inverse cannot be default implemented.

Operator.is_functional
`Operator.`**`is_functional`**
`True` if the this operator's range is a *`Field`*.

Operator.is_linear
`Operator.`**`is_linear`**
`True` if this operator is linear.

Operator.range
`Operator.`**`range`**
The range of this operator.

Methods

`__call__`(x[, out])	Return `op(x)`.
`__eq__`	Return self==value.
`derivative`(point)	Return the operator derivative at `point`.

Operator.__call__
`Operator.`**`__call__`**(*x*, *out=None*, **args*, ***kwargs*)
Return `op(x)`.

Implementation of the call pattern `op(x)` with the private `_call()` method and added error checking.

Parametersx : *`Operator.domain`* element

An object in the operator domain to which the operator is applied. The object is treated as immutable, hence it is not modified during evaluation.

out : *`Operator.range`* element, optional

An object in the operator range to which the result of the operator evaluation is written. The result is independent of the initial state of this object.

***args, **kwargs** : Further arguments to the function, optional

Returnselem : *`Operator.range`* element

An object in the operator range, the result of the operator evaluation. It is identical to `out` if provided.

Examples

```
>>> from odl import Rn, ScalingOperator
>>> rn = Rn(3)
```

```
>>> op = ScalingOperator(rn, 2.0)
>>> x = rn.element([1, 2, 3])
```

Out-of-place evaluation:

```
>>> op(x)
Rn(3).element([2.0, 4.0, 6.0])
```

In-place evaluation:

```
>>> y = rn.element()
>>> op(x, out=y)
Rn(3).element([2.0, 4.0, 6.0])
>>> y
Rn(3).element([2.0, 4.0, 6.0])
```

Operator.derivative

`Operator.`**`derivative`**(*point*)

Return the operator derivative at `point`.

RaisesOpNotImplementedError

If the operator is not linear, the derivative cannot be default implemented.

`__init__`(*domain*, *range*, *linear=False*)

Initialize a new instance.

Parametersdom : *Set*

The domain of this operator, i.e., the set of elements to which this operator can be applied

ran : *Set*

The range of this operator, i.e., the set this operator maps to

OperatorComp

class `odl.operator.operator.`**`OperatorComp`**(*left*, *right*, *tmp=None*)

Bases: *odl.operator.operator.Operator*

Expression type for the composition of operators.

`OperatorComp(left, right) <==> (x --> left(right(x)))`

The composition is only well-defined if `left.domain == right.range`.

Attributes

adjoint	The operator adjoint.
domain	The domain of this operator.
inverse	The operator inverse.
is_functional	`True` if the this operator's range is a *Field*.
is_linear	`True` if this operator is linear.
range	The range of this operator.

OperatorComp.adjoint

OperatorComp.**adjoint**
The operator adjoint.

The adjoint of the operator composition is the composition of the operator adjoints in reverse order:

```
OperatorComp(left, right).adjoint == OperatorComp(right.adjoint,
left.adjoint)
```

RaisesOpNotImplementedError

If any of the underlying operators are non-linear.

OperatorComp.domain
OperatorComp.**domain**
The domain of this operator.

OperatorComp.inverse
OperatorComp.**inverse**
The operator inverse.

The inverse of the operator composition is the composition of the inverses in reverse order:

```
OperatorComp(left, right).inverse ==       OperatorComp(right.inverse,
left.inverse)
```

OperatorComp.is_functional
OperatorComp.**is_functional**
`True` if the this operator's range is a *Field*.

OperatorComp.is_linear
OperatorComp.**is_linear**
`True` if this operator is linear.

OperatorComp.range
OperatorComp.**range**
The range of this operator.

Methods

__call__(x[, out])	Return `op(x)`.
__eq__	Return self==value.
_apply(x, out)	Implement `op(x, out=out)`.
_call(x)	Return `op(x)`.
derivative(point)	Return the operator derivative.

OperatorComp.__call__
OperatorComp.**__call__**(*x*, *out=None*, **args*, ***kwargs*)
Return `op(x)`.

Implementation of the call pattern `op(x)` with the private `_call()` method and added error checking.

Parametersx : *Operator.domain* element

An object in the operator domain to which the operator is applied. The object is treated as immutable, hence it is not modified during evaluation.

out : *Operator.range* element, optional

An object in the operator range to which the result of the operator evaluation is written. The result is independent of the initial state of this object.

***args, **kwargs** : Further arguments to the function, optional

Returnselem : *Operator.range* element

An object in the operator range, the result of the operator evaluation. It is identical to `out` if provided.

Examples

```
>>> from odl import Rn, ScalingOperator
>>> rn = Rn(3)
>>> op = ScalingOperator(rn, 2.0)
>>> x = rn.element([1, 2, 3])
```

Out-of-place evaluation:

```
>>> op(x)
Rn(3).element([2.0, 4.0, 6.0])
```

In-place evaluation:

```
>>> y = rn.element()
>>> op(x, out=y)
Rn(3).element([2.0, 4.0, 6.0])
>>> y
Rn(3).element([2.0, 4.0, 6.0])
```

OperatorComp._apply
`OperatorComp.`**`_apply`**(*x*, *out*)
Implement `op(x, out=out)`.

OperatorComp._call
`OperatorComp.`**`_call`**(*x*)
Return `op(x)`.

OperatorComp.derivative
`OperatorComp.`**`derivative`**(*point*)
Return the operator derivative.

The derivative of the operator composition follows the chain rule:

```
OperatorComp(left, right).derivative(point) == OperatorComp(left.derivative(right(p
right.derivative(point))
```

`__init__`(*left*, *right*, *tmp=None*)
Initialize a new *OperatorComp* instance.

Parametersleft : *Operator*

The left ("outer") operator

right : *Operator*

The right ("inner") operator. Its range must coincide with the domain of `left`.

tmp : element of the range of `right`, optional

Used to avoid the creation of a temporary when applying the operator.

OperatorLeftScalarMult

class `odl.operator.operator.`**`OperatorLeftScalarMult`**(*op*, *scalar*)

Bases: *odl.operator.operator.Operator*

Expression type for the operator left scalar multiplication.

`OperatorLeftScalarMult(op, scalar) <==> (x --> scalar * op(x))`

The scalar multiplication is well-defined only if `op.range` is a *LinearSpace*.

Attributes

adjoint	The operator adjoint.
domain	The domain of this operator.
inverse	The inverse operator.
is_functional	`True` if the this operator's range is a *Field*.
is_linear	`True` if this operator is linear.
range	The range of this operator.

OperatorLeftScalarMult.adjoint

`OperatorLeftScalarMult.`**`adjoint`**

The operator adjoint.

The adjoint of the operator scalar multiplication is the scalar multiplication of the operator adjoint:

`OperatorLeftScalarMult(op, scalar).adjoint == OperatorLeftScalarMult(op.adjoint, scalar)`

RaisesOpNotImplementedError

If the underlying operator is non-linear.

OperatorLeftScalarMult.domain

`OperatorLeftScalarMult.`**`domain`**

The domain of this operator.

OperatorLeftScalarMult.inverse

`OperatorLeftScalarMult.`**`inverse`**

The inverse operator.

The inverse of `scalar * op` is given by `op.inverse * 1/scalar` if `scalar != 0`. If `scalar == 0`, the inverse is not defined.

`OperatorLeftScalarMult(op, scalar).inverse <==> OperatorRightScalarMult(op.inverse, 1.0/scalar)`

OperatorLeftScalarMult.is_functional
`OperatorLeftScalarMult.is_functional`
`True` if the this operator's range is a *Field*.

OperatorLeftScalarMult.is_linear
`OperatorLeftScalarMult.is_linear`
`True` if this operator is linear.

OperatorLeftScalarMult.range
`OperatorLeftScalarMult.range`
The range of this operator.

Methods

__call__(x[, out])	Return `op(x)`.
__eq__	Return self==value.
_apply(x, out)	Implement `op(x, out=out)`.
_call(x)	Return Return `op(x)`.
derivative(x)	Return the derivative at `x`.

OperatorLeftScalarMult.__call__
`OperatorLeftScalarMult.__call__`(*x*, *out=None*, **args*, ***kwargs*)
Return `op(x)`.

Implementation of the call pattern `op(x)` with the private `_call()` method and added error checking.

Parametersx : *Operator.domain* element

An object in the operator domain to which the operator is applied. The object is treated as immutable, hence it is not modified during evaluation.

out : *Operator.range* element, optional

An object in the operator range to which the result of the operator evaluation is written. The result is independent of the initial state of this object.

***args, **kwargs** : Further arguments to the function, optional

Returnselem : *Operator.range* element

An object in the operator range, the result of the operator evaluation. It is identical to `out` if provided.

Examples

```
>>> from odl import Rn, ScalingOperator
>>> rn = Rn(3)
>>> op = ScalingOperator(rn, 2.0)
>>> x = rn.element([1, 2, 3])
```

Out-of-place evaluation:

```
>>> op(x)
Rn(3).element([2.0, 4.0, 6.0])
```

In-place evaluation:

```
>>> y = rn.element()
>>> op(x, out=y)
Rn(3).element([2.0, 4.0, 6.0])
>>> y
Rn(3).element([2.0, 4.0, 6.0])
```

OperatorLeftScalarMult._apply
`OperatorLeftScalarMult.`**`_apply`**(*x*, *out*)
Implement `op(x, out=out)`.

OperatorLeftScalarMult._call
`OperatorLeftScalarMult.`**`_call`**(*x*)
Return Return `op(x)`.

OperatorLeftScalarMult.derivative
`OperatorLeftScalarMult.`**`derivative`**(*x*)
Return the derivative at `x`.

Left scalar multiplication and derivative are commutative:

```
OperatorLeftScalarMult(op, scalar).derivative(x) <==>
OperatorLeftScalarMult(op.derivative(x), scalar)
```

See also:

OperatorLeftScalarMultthe result

__init__(*op*, *scalar*)
Initialize a new *OperatorLeftScalarMult* instance.

Parametersop : *Operator*

The range of `op` must be a *LinearSpace* or *Field*.

scalar : `op.range.field` element

A real or complex number, depending on the field of the range.

OperatorLeftVectorMult

class `odl.operator.operator.`**`OperatorLeftVectorMult`**(*op*, *vector*)
Bases: *odl.operator.operator.Operator*

Expression type for the operator left vector multiplication.

`OperatorLeftVectorMult(op, vector)(x) <==> vector * op(x)`

The scalar multiplication is well-defined only if `op.range` is a `vector.space.field`.

Attributes

adjoint	The operator adjoint.

Continued on next page

Table 2.77 – continued from previous page

domain	The domain of this operator.
inverse	Return the operator inverse.
is_functional	`True` if the this operator's range is a *Field*.
is_linear	`True` if this operator is linear.
range	The range of this operator.

OperatorLeftVectorMult.adjoint

`OperatorLeftVectorMult.`**`adjoint`**

The operator adjoint.

The adjoint of the operator vector multiplication is the vector multiplication of the operator adjoint:

```
OperatorLeftVectorMult(op, vector).adjoint == OperatorRightVectorMult(op.adjoint,
vector)
```

```
(x * A)^T = A^T * x
```

RaisesOpNotImplementedError

If the underlying operator is non-linear.

OperatorLeftVectorMult.domain

`OperatorLeftVectorMult.`**`domain`**

The domain of this operator.

OperatorLeftVectorMult.inverse

`OperatorLeftVectorMult.`**`inverse`**

Return the operator inverse.

RaisesOpNotImplementedError

Since the inverse cannot be default implemented.

OperatorLeftVectorMult.is_functional

`OperatorLeftVectorMult.`**`is_functional`**

`True` if the this operator's range is a *Field*.

OperatorLeftVectorMult.is_linear

`OperatorLeftVectorMult.`**`is_linear`**

`True` if this operator is linear.

OperatorLeftVectorMult.range

`OperatorLeftVectorMult.`**`range`**

The range of this operator.

Methods

__call__(x[, out])	Return `op(x)`.
__eq__	Return self==value.

Continued on next page

Table 2.78 – continued from previous page

_apply(x, out)	Implement `op(x, out=out)`.
_call(x)	Return `op(x)`.
derivative(x)	Return the derivative at `x`.

OperatorLeftVectorMult.__call__

`OperatorLeftVectorMult.__call__`(*x*, *out=None*, **args*, ***kwargs*)

Return `op(x)`.

Implementation of the call pattern `op(x)` with the private `_call()` method and added error checking.

Parametersx : *Operator.domain* element

An object in the operator domain to which the operator is applied. The object is treated as immutable, hence it is not modified during evaluation.

out : *Operator.range* element, optional

An object in the operator range to which the result of the operator evaluation is written. The result is independent of the initial state of this object.

***args, **kwargs** : Further arguments to the function, optional

Returnselem : *Operator.range* element

An object in the operator range, the result of the operator evaluation. It is identical to `out` if provided.

Examples

```
>>> from odl import Rn, ScalingOperator
>>> rn = Rn(3)
>>> op = ScalingOperator(rn, 2.0)
>>> x = rn.element([1, 2, 3])
```

Out-of-place evaluation:

```
>>> op(x)
Rn(3).element([2.0, 4.0, 6.0])
```

In-place evaluation:

```
>>> y = rn.element()
>>> op(x, out=y)
Rn(3).element([2.0, 4.0, 6.0])
>>> y
Rn(3).element([2.0, 4.0, 6.0])
```

OperatorLeftVectorMult._apply

`OperatorLeftVectorMult._apply`(*x*, *out*)

Implement `op(x, out=out)`.

OperatorLeftVectorMult._call

`OperatorLeftVectorMult._call`(*x*)

Return `op(x)`.

OperatorLeftVectorMult.derivative
`OperatorLeftVectorMult.derivative(x)`
Return the derivative at `x`.

Left scalar multiplication and derivative are commutative:

```
OperatorLeftVectorMult(op, vector).derivative(x) <==>
OperatorLeftVectorMult(op.derivative(x), vector)
```

See also:

OperatorLeftVectorMult the result

`__init__(op, vector)`
Initialize a new *OperatorLeftVectorMult* instance.

Parametersop : *Operator*

The range of `op` must be a *LinearSpace*.

vector : *LinearSpaceVector* in `op.range`

The vector to multiply by

OperatorPointwiseProduct

class `odl.operator.operator.OperatorPointwiseProduct(op1, op2)`
Bases: *odl.operator.operator.Operator*

Expression type for the pointwise operator mulitplication.

```
OperatorPointwiseProduct(op1, op2) <==> (x --> op1(x) * op2(x))
```

Attributes

adjoint	The operator adjoint (abstract).
domain	The domain of this operator.
inverse	Return the operator inverse.
is_functional	`True` if the this operator's range is a *Field*.
is_linear	`True` if this operator is linear.
range	The range of this operator.

OperatorPointwiseProduct.adjoint
`OperatorPointwiseProduct.adjoint`
The operator adjoint (abstract).

RaisesOpNotImplementedError

Since the adjoint cannot be default implemented.

OperatorPointwiseProduct.domain
`OperatorPointwiseProduct.domain`
The domain of this operator.

OperatorPointwiseProduct.inverse

OperatorPointwiseProduct.**inverse**
Return the operator inverse.

RaisesOpNotImplementedError

Since the inverse cannot be default implemented.

OperatorPointwiseProduct.is_functional
OperatorPointwiseProduct.**is_functional**
`True` if the this operator's range is a *Field*.

OperatorPointwiseProduct.is_linear
OperatorPointwiseProduct.**is_linear**
`True` if this operator is linear.

OperatorPointwiseProduct.range
OperatorPointwiseProduct.**range**
The range of this operator.

Methods

__call__(x[, out])	Return `op(x)`.
__eq__	Return self==value.
_apply(x, out)	Implement `op(x, out=out)`.
_call(x)	Return `op(x)`.
derivative(point)	Return the operator derivative at `point`.

OperatorPointwiseProduct.__call__
OperatorPointwiseProduct.**__call__**(*x*, *out=None*, **args*, ***kwargs*)
Return `op(x)`.

Implementation of the call pattern `op(x)` with the private `_call()` method and added error checking.

Parametersx : *Operator.domain* element

An object in the operator domain to which the operator is applied. The object is treated as immutable, hence it is not modified during evaluation.

out : *Operator.range* element, optional

An object in the operator range to which the result of the operator evaluation is written. The result is independent of the initial state of this object.

***args, **kwargs** : Further arguments to the function, optional

Returnselem : *Operator.range* element

An object in the operator range, the result of the operator evaluation. It is identical to `out` if provided.

Examples

```
>>> from odl import Rn, ScalingOperator
>>> rn = Rn(3)
>>> op = ScalingOperator(rn, 2.0)
>>> x = rn.element([1, 2, 3])
```

Out-of-place evaluation:

```
>>> op(x)
Rn(3).element([2.0, 4.0, 6.0])
```

In-place evaluation:

```
>>> y = rn.element()
>>> op(x, out=y)
Rn(3).element([2.0, 4.0, 6.0])
>>> y
Rn(3).element([2.0, 4.0, 6.0])
```

OperatorPointwiseProduct._apply
`OperatorPointwiseProduct.`**`_apply`**(*x*, *out*)
Implement `op(x, out=out)`.

OperatorPointwiseProduct._call
`OperatorPointwiseProduct.`**`_call`**(*x*)
Return `op(x)`.

OperatorPointwiseProduct.derivative
`OperatorPointwiseProduct.`**`derivative`**(*point*)
Return the operator derivative at `point`.

RaisesOpNotImplementedError

If the operator is not linear, the derivative cannot be default implemented.

`__init__`(*op1*, *op2*)
Initialize a new instance.

Parametersop1 : *Operator*

The first factor

op2 : *Operator*

The second factor. Must have the same domain and range as `op1`.

OperatorRightScalarMult

class `odl.operator.operator.`**`OperatorRightScalarMult`**(*op*, *scalar*, *tmp=None*)
Bases: *odl.operator.operator.Operator*

Expression type for the operator right scalar multiplication.

`OperatorRightScalarMult(op, scalar) <==> (x --> op(scalar * x))`

The scalar multiplication is well-defined only if `op.domain` is a *LinearSpace*.

Attributes

adjoint	The operator adjoint.
domain	The domain of this operator.
inverse	The inverse operator.
is_functional	`True` if the this operator's range is a *Field*.
is_linear	`True` if this operator is linear.
range	The range of this operator.

OperatorRightScalarMult.adjoint

`OperatorRightScalarMult.`**`adjoint`**

The operator adjoint.

The adjoint of the operator scalar multiplication is the scalar multiplication of the operator adjoint:

```
OperatorLeftScalarMult(op, scalar).adjoint == OperatorLeftScalarMult(op.adjoint,
scalar)
```

RaisesOpNotImplementedError

If the underlying operator is non-linear.

OperatorRightScalarMult.domain

`OperatorRightScalarMult.`**`domain`**

The domain of this operator.

OperatorRightScalarMult.inverse

`OperatorRightScalarMult.`**`inverse`**

The inverse operator.

The inverse of `op * scalar` is given by `1/scalar * op.inverse` if `scalar != 0`. If `scalar == 0`, the inverse is not defined.

```
OperatorRightScalarMult(op, scalar).inverse <==> OperatorLeftScalarMult(op.inverse,
1.0/scalar)
```

OperatorRightScalarMult.is_functional

`OperatorRightScalarMult.`**`is_functional`**

`True` if the this operator's range is a *Field*.

OperatorRightScalarMult.is_linear

`OperatorRightScalarMult.`**`is_linear`**

`True` if this operator is linear.

OperatorRightScalarMult.range

`OperatorRightScalarMult.`**`range`**

The range of this operator.

Methods

__call__(x[, out])	Return `op(x)`.
__eq__	Return self==value.
_apply(x, out)	Implement `self(x, out=out)`.
_call(x)	Return `self(x)`.
derivative(x)	Return the derivative at `x`.

OperatorRightScalarMult.__call__

`OperatorRightScalarMult.`**`__call__`**(*x*, *out=None*, **args*, ***kwargs*)

Return `op(x)`.

Implementation of the call pattern `op(x)` with the private `_call()` method and added error checking.

Parametersx : *Operator.domain* element

An object in the operator domain to which the operator is applied. The object is treated as immutable, hence it is not modified during evaluation.

out : *Operator.range* element, optional

An object in the operator range to which the result of the operator evaluation is written. The result is independent of the initial state of this object.

***args, **kwargs** : Further arguments to the function, optional

Returnselem : *Operator.range* element

An object in the operator range, the result of the operator evaluation. It is identical to `out` if provided.

Examples

```
>>> from odl import Rn, ScalingOperator
>>> rn = Rn(3)
>>> op = ScalingOperator(rn, 2.0)
>>> x = rn.element([1, 2, 3])
```

Out-of-place evaluation:

```
>>> op(x)
Rn(3).element([2.0, 4.0, 6.0])
```

In-place evaluation:

```
>>> y = rn.element()
>>> op(x, out=y)
Rn(3).element([2.0, 4.0, 6.0])
>>> y
Rn(3).element([2.0, 4.0, 6.0])
```

OperatorRightScalarMult._apply

`OperatorRightScalarMult.`**`_apply`**(*x*, *out*)

Implement `self(x, out=out)`.

OperatorRightScalarMult._call

`OperatorRightScalarMult.`**`_call`**(*x*)

Return `self(x)`.

OperatorRightScalarMult.derivative
`OperatorRightScalarMult.`**`derivative`**`(x)`
Return the derivative at `x`.

The derivative of the right scalar operator multiplication follows the chain rule:

```
OperatorRightScalarMult(op, scalar).derivative(x) <==>
OperatorLeftScalarMult(op.derivative(scalar * x), scalar)
```

`__init__``(op, scalar, tmp=None)`
Initialize a new *OperatorLeftScalarMult* instance.

Parametersop : *Operator*

The domain of `op` must be a *LinearSpace* or *Field*.

scalar : `op.range.field` element

A real or complex number, depending on the field of the operator domain.

tmp : domain element, optional

Used to avoid the creation of a temporary when applying the operator.

OperatorRightVectorMult

class `odl.operator.operator.`**`OperatorRightVectorMult`**`(op, vector)`
Bases: *odl.operator.operator.Operator*

Expression type for the operator right vector multiplication.

```
OperatorRightVectorMult(op, vector)(x) <==> op(vector * x)
```

The scalar multiplication is well-defined only if `vector in op.domain == True`.

Attributes

adjoint	The operator adjoint.
domain	The domain of this operator.
inverse	Return the operator inverse.
is_functional	`True` if the this operator's range is a *Field*.
is_linear	`True` if this operator is linear.
range	The range of this operator.

OperatorRightVectorMult.adjoint
`OperatorRightVectorMult.`**`adjoint`**
The operator adjoint.

The adjoint of the operator vector multiplication is the vector multiplication of the operator adjoint:

```
OperatorRightVectorMult(op, vector).adjoint == OperatorLeftVectorMult(op.adjoint,
vector)
```

```
(A x)^T = x * A^T
```

RaisesOpNotImplementedError

If the underlying operator is non-linear.

OperatorRightVectorMult.domain
`OperatorRightVectorMult.`**`domain`**
The domain of this operator.

OperatorRightVectorMult.inverse
`OperatorRightVectorMult.`**`inverse`**
Return the operator inverse.

RaisesOpNotImplementedError

Since the inverse cannot be default implemented.

OperatorRightVectorMult.is_functional
`OperatorRightVectorMult.`**`is_functional`**
`True` if the this operator's range is a *`Field`*.

OperatorRightVectorMult.is_linear
`OperatorRightVectorMult.`**`is_linear`**
`True` if this operator is linear.

OperatorRightVectorMult.range
`OperatorRightVectorMult.`**`range`**
The range of this operator.

Methods

`__call__`(x[, out])	Return `op(x)`.
`__eq__`	Return self==value.
`_apply`(x, out)	Implement `op(x, out=out)`.
`_call`(x)	Return `op(x)`.
`derivative`(x)	Return the derivative at `x`.

OperatorRightVectorMult.__call__
`OperatorRightVectorMult.`**`__call__`**(*x*, *out=None*, **args*, ***kwargs*)
Return `op(x)`.

Implementation of the call pattern `op(x)` with the private `_call()` method and added error checking.

Parametersx : *`Operator.domain`* element

An object in the operator domain to which the operator is applied. The object is treated as immutable, hence it is not modified during evaluation.

out : *`Operator.range`* element, optional

An object in the operator range to which the result of the operator evaluation is written. The result is independent of the initial state of this object.

***args, **kwargs** : Further arguments to the function, optional

Returnselem : *`Operator.range`* element

An object in the operator range, the result of the operator evaluation. It is identical to `out` if provided.

Examples

```
>>> from odl import Rn, ScalingOperator
>>> rn = Rn(3)
>>> op = ScalingOperator(rn, 2.0)
>>> x = rn.element([1, 2, 3])
```

Out-of-place evaluation:

```
>>> op(x)
Rn(3).element([2.0, 4.0, 6.0])
```

In-place evaluation:

```
>>> y = rn.element()
>>> op(x, out=y)
Rn(3).element([2.0, 4.0, 6.0])
>>> y
Rn(3).element([2.0, 4.0, 6.0])
```

OperatorRightVectorMult._apply
`OperatorRightVectorMult.`**`_apply`**(*x*, *out*)
Implement `op(x, out=out)`.

OperatorRightVectorMult._call
`OperatorRightVectorMult.`**`_call`**(*x*)
Return `op(x)`.

OperatorRightVectorMult.derivative
`OperatorRightVectorMult.`**`derivative`**(*x*)
Return the derivative at `x`.

Left vector multiplication and derivative are commutative:

```
OperatorRightVectorMult(op, vector).derivative(x) <==>
OperatorRightVectorMult(op.derivative(x), vector)
```

See also:

OperatorRightVectorMult the result

`__init__`(*op*, *vector*)
Initialize a new *OperatorRightVectorMult* instance.

Parameters
op : *Operator*
The domain of `op` must be a `vector.space`.
vector : *LinearSpaceVector* in `op.domain`
The vector to multiply by

OperatorSum

class `odl.operator.operator.`**`OperatorSum`**(*op1*, *op2*, *tmp_ran=None*, *tmp_dom=None*)
Bases: *odl.operator.operator.Operator*

Expression type for the sum of operators.

`OperatorSum(op1, op2) <==> (x --> op1(x) + op2(x))`

The sum is only well-defined for *Operator* instances where *Operator.range* is a *LinearSpace*.

Attributes

adjoint	The operator adjoint.
domain	The domain of this operator.
inverse	Return the operator inverse.
is_functional	`True` if the this operator's range is a *Field*.
is_linear	`True` if this operator is linear.
range	The range of this operator.

OperatorSum.adjoint

`OperatorSum.`**`adjoint`**

The operator adjoint.

The adjoint of the operator sum is the sum of the operator adjoints:

```
OperatorSum(op1, op2).adjoint == OperatorSum(op1.adjoint, op2.adjoint)
```

RaisesOpNotImplementedError

If either of the underlying operators are non-linear.

OperatorSum.domain

`OperatorSum.`**`domain`**

The domain of this operator.

OperatorSum.inverse

`OperatorSum.`**`inverse`**

Return the operator inverse.

RaisesOpNotImplementedError

Since the inverse cannot be default implemented.

OperatorSum.is_functional

`OperatorSum.`**`is_functional`**

`True` if the this operator's range is a *Field*.

OperatorSum.is_linear

`OperatorSum.`**`is_linear`**

`True` if this operator is linear.

OperatorSum.range

`OperatorSum.`**`range`**

The range of this operator.

Methods

__call__(x[, out])	Return `op(x)`.
__eq__	Return self==value.
_apply(x, out)	Implement `op(x, out=out)`.
_call(x)	Return `op(x)`.
derivative(x)	Return the operator derivative at `x`.

OperatorSum.__call__

`OperatorSum.__call__`(*x*, *out=None*, **args*, ***kwargs*)

Return `op(x)`.

Implementation of the call pattern `op(x)` with the private `_call()` method and added error checking.

Parametersx : *Operator.domain* element

An object in the operator domain to which the operator is applied. The object is treated as immutable, hence it is not modified during evaluation.

out : *Operator.range* element, optional

An object in the operator range to which the result of the operator evaluation is written. The result is independent of the initial state of this object.

***args, **kwargs** : Further arguments to the function, optional

Returnselem : *Operator.range* element

An object in the operator range, the result of the operator evaluation. It is identical to `out` if provided.

Examples

```
>>> from odl import Rn, ScalingOperator
>>> rn = Rn(3)
>>> op = ScalingOperator(rn, 2.0)
>>> x = rn.element([1, 2, 3])
```

Out-of-place evaluation:

```
>>> op(x)
Rn(3).element([2.0, 4.0, 6.0])
```

In-place evaluation:

```
>>> y = rn.element()
>>> op(x, out=y)
Rn(3).element([2.0, 4.0, 6.0])
>>> y
Rn(3).element([2.0, 4.0, 6.0])
```

OperatorSum._apply

`OperatorSum._apply`(*x*, *out*)

Implement `op(x, out=out)`.

Examples

```
>>> from odl import Rn, IdentityOperator
>>> r3 = Rn(3)
>>> op = IdentityOperator(r3)
>>> x = r3.element([1, 2, 3])
>>> out = r3.element()
>>> OperatorSum(op, op)(x, out)
Rn(3).element([2.0, 4.0, 6.0])
>>> out
Rn(3).element([2.0, 4.0, 6.0])
```

OperatorSum._call

`OperatorSum.__call__(x)`

Return `op(x)`.

Examples

```
>>> from odl import Rn, ScalingOperator
>>> r3 = Rn(3)
>>> A = ScalingOperator(r3, 3.0)
>>> B = ScalingOperator(r3, -1.0)
>>> C = OperatorSum(A, B)
>>> C(r3.element([1, 2, 3]))
Rn(3).element([2.0, 4.0, 6.0])
```

OperatorSum.derivative

`OperatorSum.derivative(x)`

Return the operator derivative at `x`.

TODO: finish doc

The derivative of a sum of two operators is equal to the sum of the derivatives.

`__init__(op1, op2, tmp_ran=None, tmp_dom=None)`

Initialize a new instance.

Parametersop1 : *Operator*

The first summand. Its *Operator.range* must be a *LinearSpace* or *Field*.

op2 : *Operator*

The second summand. Must have the same *Operator.domain* and *Operator.range* as `op1`.

tmp_ran : *Operator.range* element, optional

Used to avoid the creation of a temporary when applying the operator.

tmp_dom : *Operator.domain* element, optional

Used to avoid the creation of a temporary when applying the operator adjoint.

Functions

simple_operator([call, apply, inv, deriv, ...])	Create a simple operator.

simple_operator

odl.operator.operator.**simple_operator**(*call=None*, *apply=None*, *inv=None*, *deriv=None*, *dom=None*, *ran=None*, *linear=False*)

Create a simple operator.

Mostly intended for simple prototyping rather than final use.

Parameters

call : `callable`

A function taking one argument and returning the result. It will be used for the operator call pattern `out = op(x)`.

apply : `callable`

A function taking two arguments. It will be used for the operator apply pattern Implement `op(x, out=out)`. Return value is assumed to be `None` and is ignored.

inv : *Operator*, optional

The operator inverse

deriv : *Operator*, optional

The operator derivative, linear

dom : *Set*, optional

The domain of the operator Default: *UniversalSpace* if linear, else *UniversalSet*

ran : *Set*, optional

The range of the operator Default: *UniversalSpace* if linear, else *UniversalSet*

linear : `bool`, optional

`True` if the operator is linear Default: `False`

Returns

op : *Operator*

An operator with the provided attributes and methods.

Notes

It suffices to supply one of the functions `call` and `apply`. If `dom` is a *LinearSpace*, a default implementation of the respective other method is automatically provided; if not, a *OpNotImplementedError* is raised when the other method is called.

Examples

```
>>> A = simple_operator(lambda x: 3*x)
>>> A(5)
15
```

oputils

Usefull utility functions on discrete spaces (i.e., either Rn/Cn or discretized function spaces), for example obtaining a matrix representation of an operator.

Functions

matrix_representation(op)	Returns a matrix representation of a linear operator.
power_method_opnorm(op, niter[, xstart])	Estimate the operator norm with the power method.

matrix_representation

odl.operator.oputils.**matrix_representation**(*op*)

Returns a matrix representation of a linear operator.

Parameters
op : `Operator`

The linear operator of which one wants a matrix representation.

Returns
matrix : `numpy.ndarray`

The matrix representation of the operator.

Notes

The algorithm works by letting the operator act on all unit vectors, and stacking the output as a matrix.

power_method_opnorm

odl.operator.oputils.**power_method_opnorm**(*op*, *niter*, *xstart=None*)

Estimate the operator norm with the power method.

Parameters
op : *Operator*

Operator whose norm is to be estimated. If its *Operator.range* range does not coincide with its *Operator.domain*, an *Operator.adjoint* must be defined (which implies that the operator must be linear).

niter : positive `int`

Number of iterations to perform

xstart : *Operator.domain* element, optional

Starting point of the iteration. By default, the `one` element of the *Operator.domain* is used.

Returns
est_norm : `float`

The estimated operator norm

pspace_ops

Default operators defined on any *ProductSpace*.

Classes

ComponentProjection(space, index)	Projection onto the subspace identified by an index.
ComponentProjectionAdjoint(space, index)	Adjoint operator to *ComponentProjection*.
ProductSpaceOperator(operators[, dom, ran])	A "matrix of operators" on product spaces.

ComponentProjection

class odl.operator.pspace_ops.**ComponentProjection**(*space*, *index*)

Bases: *odl.operator.operator.Operator*

Projection onto the subspace identified by an index.

For a product space $\mathcal{X} = \mathcal{X}_1 \times \cdots \times \mathcal{X}_n$, the component projection

$$\mathcal{P}_i : \mathcal{X} \to \mathcal{X}_i$$

is given by $\mathcal{P}_i(x) = x_i$ for an element $x = (x_1, \ldots, x_n) \in \mathcal{X}$.

More generally, for an index set $I \subset \{1, \ldots, n\}$, the projection operator $\mathcal{P}_I$ is defined by $\mathcal{P}_I(x) = (x_i)_{i \in I}$.

Note that this is a special case of a product space operator where the "operator matrix" has only one row and contains only identity operators.

Attributes

adjoint	Return the adjoint operator.
domain	The domain of this operator.
index	Index of the subspace.
inverse	Return the operator inverse.
is_functional	`True` if the this operator's range is a *Field*.
is_linear	`True` if this operator is linear.
range	The range of this operator.

ComponentProjection.adjoint

ComponentProjection.**adjoint**

Return the adjoint operator.

The adjoint is given by extending along *ComponentProjection.index*, and setting zero along the others.

See also:

ComponentProjectionAdjoint

ComponentProjection.domain

ComponentProjection.**domain**

The domain of this operator.

ComponentProjection.index

ComponentProjection.**index**

Index of the subspace.

ComponentProjection.inverse
`ComponentProjection.`**`inverse`**
Return the operator inverse.

RaisesOpNotImplementedError

Since the inverse cannot be default implemented.

ComponentProjection.is_functional
`ComponentProjection.`**`is_functional`**
`True` if the this operator's range is a *`Field`*.

ComponentProjection.is_linear
`ComponentProjection.`**`is_linear`**
`True` if this operator is linear.

ComponentProjection.range
`ComponentProjection.`**`range`**
The range of this operator.

Methods

`__eq__`	Return self==value.
`_apply`(x, out)	Project x onto subspace in-place.
`_call`(x)	Project x onto subspace out-of-place.

ComponentProjection._apply
`ComponentProjection.`**`_apply`**(*x*, *out*)
Project x onto subspace in-place.

See also:

`ComponentProjection._call`

ComponentProjection._call
`ComponentProjection.`**`_call`**(*x*)
Project x onto subspace out-of-place.

Parametersx : domain element

input vector to be projected

Returnsout : range element

Projection of x onto subspace

Examples

```
>>> import odl
>>> r1 = odl.Rn(1)
>>> r2 = odl.Rn(2)
>>> r3 = odl.Rn(3)
```

```
>>> X = odl.ProductSpace(r1, r2, r3)
>>> x = X.element([[1], [2, 3], [4, 5, 6]])
```

Projection on n-th component

```
>>> proj = odl.ComponentProjection(X, 0)
>>> proj(x)
Rn(1).element([1.0])
```

Projection on sub-space

```
>>> proj = odl.ComponentProjection(X, [0, 2])
>>> proj(x)
ProductSpace(Rn(1), Rn(3)).element([
    [1.0],
    [4.0, 5.0, 6.0]
])
```

__init__ (*space*, *index*)

Parameters **space** : *ProductSpace*

The space to project from

index : `int`, `slice`, or `iterable` [int]

The indices defining the subspace. If `index` is not and `int`, the *Operator.range* of this operator is also a *ProductSpace*.

Examples

```
>>> import odl
>>> r1 = odl.Rn(1)
>>> r2 = odl.Rn(2)
>>> r3 = odl.Rn(3)
>>> X = odl.ProductSpace(r1, r2, r3)
```

Projection on n-th component

```
>>> proj = odl.ComponentProjection(X, 0)
>>> proj.range
Rn(1)
```

Projection on sub-space

```
>>> proj = odl.ComponentProjection(X, [0, 2])
>>> proj.range
ProductSpace(Rn(1), Rn(3))
```

ComponentProjectionAdjoint

class `odl.operator.pspace_ops.`**ComponentProjectionAdjoint** (*space*, *index*)

Bases: *odl.operator.operator.Operator*

Adjoint operator to *ComponentProjection*.

As a special case of the adjoint of a *ProductSpaceOperator*, this operator is given as a column vector of identity operators and zero operators, with the identities placed in the positions defined by *ComponentProjectionAdjoint.index*.

In weighted product spaces, the adjoint needs to take the weightings into account. This is currently not supported.

Attributes

adjoint	The adjoint operator.
domain	The domain of this operator.
index	Index of the subspace.
inverse	Return the operator inverse.
is_functional	`True` if the this operator's range is a *Field*.
is_linear	`True` if this operator is linear.
range	The range of this operator.

ComponentProjectionAdjoint.adjoint
`ComponentProjectionAdjoint.`**`adjoint`**
The adjoint operator.

The adjoint is given by the *ComponentProjection* related to this operator's *index*.

See also:

ComponentProjection

ComponentProjectionAdjoint.domain
`ComponentProjectionAdjoint.`**`domain`**
The domain of this operator.

ComponentProjectionAdjoint.index
`ComponentProjectionAdjoint.`**`index`**
Index of the subspace.

ComponentProjectionAdjoint.inverse
`ComponentProjectionAdjoint.`**`inverse`**
Return the operator inverse.

RaisesOpNotImplementedError

Since the inverse cannot be default implemented.

ComponentProjectionAdjoint.is_functional
`ComponentProjectionAdjoint.`**`is_functional`**
`True` if the this operator's range is a *Field*.

ComponentProjectionAdjoint.is_linear
`ComponentProjectionAdjoint.`**`is_linear`**
`True` if this operator is linear.

ComponentProjectionAdjoint.range
`ComponentProjectionAdjoint.`**`range`**
The range of this operator.

Methods

`__eq__`	Return self==value.
`_apply`(x, out)	Evaluate this operator in-place.
`_call`(x)	Evaluate this operator out-of-place.

ComponentProjectionAdjoint._apply

`ComponentProjectionAdjoint.`**`_apply`**(*x*, *out*)

Evaluate this operator in-place.

Extend x from the subspace related to *`index`*.

See also:

`ComponentProjectionAdjoint._call`

ComponentProjectionAdjoint._call

`ComponentProjectionAdjoint.`**`_call`**(*x*)

Evaluate this operator out-of-place.

Extend x from the subspace related to *`index`*.

Parametersx : domain element

Input vector to be projected

Returnsout : range element

Projection of x to superspace

Examples

```
>>> import odl
>>> r1 = odl.Rn(1)
>>> r2 = odl.Rn(2)
>>> r3 = odl.Rn(3)
>>> X = odl.ProductSpace(r1, r2, r3)
>>> x = X.element([[1], [2, 3], [4, 5, 6]])
```

Projection on n-th component

```
>>> proj = odl.ComponentProjectionAdjoint(X, 0)
>>> proj(x[0])
ProductSpace(Rn(1), Rn(2), Rn(3)).element([
    [1.0],
    [0.0, 0.0],
    [0.0, 0.0, 0.0]
])
```

Projection on sub-space

```
>>> proj = odl.ComponentProjectionAdjoint(X, [0, 2])
>>> proj(x[0, 2])
ProductSpace(Rn(1), Rn(2), Rn(3)).element([
    [1.0],
    [0.0, 0.0],
```

```
            [4.0, 5.0, 6.0]
        ])
```

__init__ (*space*, *index*)

Initialize a new instance

Parameters **space** : *ProductSpace*

The space to project to

index : `int`, `slice`, or `iterable` [int]

The indexes to project from

Examples

```
>>> import odl
>>> r1 = odl.Rn(1)
>>> r2 = odl.Rn(2)
>>> r3 = odl.Rn(3)
>>> X = odl.ProductSpace(r1, r2, r3)
```

Projection on n-th component

```
>>> proj = odl.ComponentProjectionAdjoint(X, 0)
>>> proj.domain
Rn(1)
```

Projection on sub-space

```
>>> proj = odl.ComponentProjectionAdjoint(X, [0, 2])
>>> proj.domain
ProductSpace(Rn(1), Rn(3))
```

ProductSpaceOperator

class `odl.operator.pspace_ops.`**ProductSpaceOperator**(*operators*, *dom=None*, *ran=None*)

Bases: *odl.operator.operator.Operator*

A "matrix of operators" on product spaces.

This is intended for the case where an operator can be decomposed as a linear combination of "sub-operators", e.g.

$$\begin{pmatrix} A & B & 0 \\ 0 & C & 0 \\ 0 & 0 & D \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} A(x) + B(y) \\ C(y) \\ D(z) \end{pmatrix}$$

Mathematically, a *ProductSpaceOperator* is an operator

$$\mathcal{A} : \mathcal{X} \to \mathcal{Y}$$

between product spaces $\mathcal{X} = \mathcal{X}_1 \times \cdots \times \mathcal{X}_m$ and $\mathcal{Y} = \mathcal{Y}_1 \times \cdots \times \mathcal{Y}_n$ which can be written in the form

$$\mathcal{A} = (\mathcal{A}_{ij})_{i,j}, \quad i = 1, \ldots, n, \ j = 1, \ldots, m$$

with *component operators* $\mathcal{A}_{ij} : \mathcal{X}_j \to \mathcal{Y}_i$.

Its action on a vector $x = (x_1, \ldots, x_m)$ is defined as the matrix multiplication

$$[\mathcal{A}(x)]_i = \sum_{j=1}^{m} \mathcal{A}_{ij}(x_j).$$

Notes

In many cases it is of interest to have an operator from a *ProductSpace* to any *LinearSpace*. It that case this operator can be used with a slight modification, simply run

`prod_op = ProductSpaceOperator(prod_space, ProductSpace(linear_space))`

The same can be done for operators *LinearSpace* -> *ProductSpace*

`prod_op = ProductSpaceOperator(ProductSpace(linear_space), prod_space)`

Attributes

adjoint	Adjoint of the product space operator.
domain	The domain of this operator.
inverse	Return the operator inverse.
is_functional	`True` if the this operator's range is a *Field*.
is_linear	`True` if this operator is linear.
range	The range of this operator.

ProductSpaceOperator.adjoint

`ProductSpaceOperator.`**`adjoint`**

Adjoint of the product space operator.

The adjoint is given by taking the transpose of the matrix and the adjoint of each component operator.

In weighted product spaces, the adjoint needs to take the weightings into account. This is currently not supported.

Returnsadjoint : *ProductSpaceOperator*

The adjoint

Examples

```
>>> import odl
>>> r3 = odl.Rn(3)
>>> X = odl.ProductSpace(r3, r3)
>>> I = odl.IdentityOperator(r3)
>>> x = X.element([[1, 2, 3], [4, 5, 6]])
```

Matrix is transposed:

```
>>> prod_op = ProductSpaceOperator([[0, I], [0, 0]],
...                                dom=X, ran=X)
>>> prod_op(x)
ProductSpace(Rn(3), 2).element([
    [4.0, 5.0, 6.0],
    [0.0, 0.0, 0.0]
])
>>> prod_op.adjoint(x)
ProductSpace(Rn(3), 2).element([
    [0.0, 0.0, 0.0],
    [1.0, 2.0, 3.0]
])
```

ProductSpaceOperator.domain
ProductSpaceOperator.**domain**
The domain of this operator.

ProductSpaceOperator.inverse
ProductSpaceOperator.**inverse**
Return the operator inverse.

RaisesOpNotImplementedError

Since the inverse cannot be default implemented.

ProductSpaceOperator.is_functional
ProductSpaceOperator.**is_functional**
`True` if the this operator's range is a *Field*.

ProductSpaceOperator.is_linear
ProductSpaceOperator.**is_linear**
`True` if this operator is linear.

ProductSpaceOperator.range
ProductSpaceOperator.**range**
The range of this operator.

Methods

__eq__	Return self==value.
_apply(x, out)	Call the ProductSpace operators in-place.
_call(x)	Call the ProductSpace operators.

ProductSpaceOperator._apply
ProductSpaceOperator.**_apply**(*x*, *out*)
Call the ProductSpace operators in-place.

Parametersx : domain element

input vector to be evaluated

out : range element

output vector to write result to

ReturnsNone

Examples

See *_call*

ProductSpaceOperator._call
ProductSpaceOperator.**_call**(*x*)
Call the ProductSpace operators.

Parametersx : domain element

Input vector to be evaluated

Returnsout : range element

Result of the evaluation

Examples

```
>>> import odl
>>> r3 = odl.Rn(3)
>>> X = odl.ProductSpace(r3, r3)
>>> I = odl.IdentityOperator(r3)
>>> x = X.element([[1, 2, 3], [4, 5, 6]])
```

Sum of elements:

```
>>> prod_op = ProductSpaceOperator([I, I])
>>> prod_op(x)
ProductSpace(Rn(3), 1).element([
    [5.0, 7.0, 9.0]
])
```

Diagonal operator – 0 or `None` means ignore, or the implicit zero operator:

```
>>> prod_op = ProductSpaceOperator([[I, 0], [0, I]])
>>> prod_op(x)
ProductSpace(Rn(3), 2).element([
    [1.0, 2.0, 3.0],
    [4.0, 5.0, 6.0]
])
```

Complicated combinations:

```
>>> prod_op = ProductSpaceOperator([[I, I], [I, 0]])
>>> prod_op(x)
ProductSpace(Rn(3), 2).element([
    [5.0, 7.0, 9.0],
    [1.0, 2.0, 3.0]
])
```

__init__ (*operators*, *dom=None*, *ran=None*)

Initialize a new instance.

Parametersoperators : array-like

An array of *Operator*'s

dom : *ProductSpace*

Domain of the operator. If not provided, it is tried to be inferred from the operators. This requires each **column** to contain at least one operator.

ran : *ProductSpace*

Range of the operator. If not provided, it is tried to be inferred from the operators. This requires each **row** to contain at least one operator.

Examples

```
>>> import odl
>>> r3 = odl.Rn(3)
>>> X = odl.ProductSpace(r3, r3)
>>> I = odl.IdentityOperator(r3)
```

Sum of elements

```
>>> prod_op = ProductSpaceOperator([I, I])
```

Diagonal operator, 0 or None means ignore, or the implicit zero op.

```
>>> prod_op = ProductSpaceOperator([[I, 0], [None, I]])
```

Complicated combinations also possible

```
>>> prod_op = ProductSpaceOperator([[I, I], [I, 0]])
```

2.4.4 set

Core Spaces and set support.

Modules

domain

Typical domains for inverse problems.

Classes

IntervalProd(begin, end)	An n-dimensional rectangular box.

IntervalProd

class odl.set.domain.**IntervalProd**(*begin*, *end*)

Bases: *odl.set.sets.Set*

An n-dimensional rectangular box.

An interval product is a Cartesian product of n intervals, i.e. an n-dimensional rectangular box aligned with the coordinate axes as a subset of R^n.

Attributes

area	The length of this interval.
begin	The left interval boundary/boundaries.
end	The right interval boundary/boundaries.
length	The length of this interval.

Continued on next page

Table 2.97 – continued from previous page

midpoint	The midpoint of the interval product.
ndim	The number of intervals in the product.
size	The interval length per axis.
true_ndim	The number of non-degenerate (zero-length) intervals.
volume	The 'dim'-dimensional volume of this interval product.

IntervalProd.area
IntervalProd.**area**
The length of this interval.

IntervalProd.begin
IntervalProd.**begin**
The left interval boundary/boundaries.

IntervalProd.end
IntervalProd.**end**
The right interval boundary/boundaries.

IntervalProd.length
IntervalProd.**length**
The length of this interval.

IntervalProd.midpoint
IntervalProd.**midpoint**
The midpoint of the interval product.

IntervalProd.ndim
IntervalProd.**ndim**
The number of intervals in the product.

IntervalProd.size
IntervalProd.**size**
The interval length per axis.

IntervalProd.true_ndim
IntervalProd.**true_ndim**
The number of non-degenerate (zero-length) intervals.

IntervalProd.volume
IntervalProd.**volume**
The 'dim'-dimensional volume of this interval product.

Continued on next page

Table 2.98 – continued from previous page

Methods

__contains__(other)	Return `other in self`.
__eq__(other)	Return `self == other`.
approx_contains(point, tol)	Test if a point is contained.
approx_equals(other, tol)	Test if `other` is equal to this set up to `tol`.
collapse(indices, values)	Partly collapse the interval product to single values.
contains_set(other[, tol])	Test if another set is contained.
corners([order])	The corner points in a single array.
dist(point[, ord])	Calculate the distance to a point.
element()	An arbitrary element, the midpoint.
insert(other[, index])	Insert another interval product before the given index.
max()	The maximum value in this interval product
measure([ndim])	The (Lebesgue) measure of this interval product.
min()	The minimum value in this interval product
squeeze()	Remove the degenerate dimensions.

IntervalProd.__contains__

`IntervalProd.`**`__contains__`**(*other*)

Return `other in self`.

IntervalProd.__eq__

`IntervalProd.`**`__eq__`**(*other*)

Return `self == other`.

IntervalProd.approx_contains

`IntervalProd.`**`approx_contains`**(*point*, *tol*)

Test if a point is contained.

Parameterspoint : array-like or `float`

The point to be tested. Its length must be equal to the set's dimension. In the 1d case, 'point' can be given as a `float`.

tol : `float`

The maximum allowed distance in 'inf'-norm between the point and the set. Default: 0.0

Examples

```
>>> from math import sqrt
>>> b, e = [-1, 0, 2], [-0.5, 0, 3]
>>> rbox = IntervalProd(b, e)
>>> # Numerical error
>>> rbox.approx_contains([-1 + sqrt(0.5)**2, 0., 2.9], tol=0)
False
>>> rbox.approx_contains([-1 + sqrt(0.5)**2, 0., 2.9], tol=1e-9)
True
```

IntervalProd.approx_equals

`IntervalProd.approx_equals(other, tol)`

Test if `other` is equal to this set up to `tol`.

Parametersother : `object`

The object to be tested

tol : `float`

The maximum allowed difference in 'inf'-norm between the interval endpoints.

Examples

```
>>> from math import sqrt
>>> rbox1 = IntervalProd(0, 0.5)
>>> rbox2 = IntervalProd(0, sqrt(0.5)**2)
>>> rbox1.approx_equals(rbox2, tol=0)   # Num error
False
>>> rbox1.approx_equals(rbox2, tol=1e-15)
True
```

IntervalProd.collapse

`IntervalProd.collapse(indices, values)`

Partly collapse the interval product to single values.

Note that no changes are made in-place.

Parametersindices : `int` or `tuple` of `int`

The indices of the dimensions along which to collapse

values : `float` or array-like

The values to which to collapse. Must have the same lenght as 'indcs'. Values must lie within the interval boundaries.

Returnscollapsed : *IntervalProd*

The collapsed set

Examples

```
>>> b, e = [-1, 0, 2], [-0.5, 1, 3]
>>> rbox = IntervalProd(b, e)
>>> rbox.collapse(1, 0)
Cuboid([-1.0, 0.0, 2.0], [-0.5, 0.0, 3.0])
>>> rbox.collapse([1, 2], [0, 2.5])
Cuboid([-1.0, 0.0, 2.5], [-0.5, 0.0, 2.5])
>>> rbox.collapse([1, 2], [0, 3.5])
Traceback (most recent call last):
    ...
ValueError: values [ 0.   3.5] not below the upper interval
boundaries [ 1.  3.].
```

IntervalProd.contains_set

`IntervalProd.`**`contains_set`**(*other*, *tol=0.0*)

Test if another set is contained.

Parametersother : *Set*

The set to be tested. It must implement a `min()` and a `max()` method, otherwise a `TypeError` is raised.

tol : `float`, optional

The maximum allowed distance in 'inf'-norm between the other set and this interval product. Default: 0.0

IntervalProd.corners

`IntervalProd.`**`corners`**(*order='C'*)

The corner points in a single array.

Parametersorder : {'C', 'F'}

The ordering of the axes in which the corners appear in the output. 'C' means that the first axis varies slowest and the last one fastest, vice versa in 'F' ordering.

Returnsout : `numpy.ndarray`

The size of the array is `2^m * ndim`, where `m` is the number of non-degenerate axes, i.e. the corners are stored as rows.

Examples

```
>>> rbox = IntervalProd([-1, 2, 0], [-0.5, 3, 0.5])
>>> rbox.corners()
array([[-1. ,  2. ,  0. ],
       [-1. ,  2. ,  0.5],
       [-1. ,  3. ,  0. ],
       [-1. ,  3. ,  0.5],
       [-0.5,  2. ,  0. ],
       [-0.5,  2. ,  0.5],
       [-0.5,  3. ,  0. ],
       [-0.5,  3. ,  0.5]])
>>> rbox.corners(order='F')
array([[-1. ,  2. ,  0. ],
       [-0.5,  2. ,  0. ],
       [-1. ,  3. ,  0. ],
       [-0.5,  3. ,  0. ],
       [-1. ,  2. ,  0.5],
       [-0.5,  2. ,  0.5],
       [-1. ,  3. ,  0.5],
       [-0.5,  3. ,  0.5]])
```

IntervalProd.dist

`IntervalProd.`**`dist`**(*point*, *ord=2.0*)

Calculate the distance to a point.

Parameterspoint : array-like or `float`

The point. Its length must be equal to the set's dimension. Can be a `float` in the 1d case.

ord : non-zero int or float('inf'), optional

The order of the norm (see `numpy.linalg.norm`). Default: 2.0

Examples

```
>>> b, e = [-1, 0, 2], [-0.5, 0, 3]
>>> rbox = IntervalProd(b, e)
>>> rbox.dist([-5, 3, 2])
5.0
>>> rbox.dist([-5, 3, 2], ord=float('inf'))
4.0
```

IntervalProd.element

`IntervalProd.`**`element`**`()`

An arbitrary element, the midpoint.

IntervalProd.insert

`IntervalProd.`**`insert`**`(`*other*, *index=None*`)`

Insert another interval product before the given index.

The given interval product (`ndim=m`) is inserted into the current one (`ndim=n`) before the given index, resulting in a new interval product with `n+m` dimensions.

No changes are made in-place.

Parametersother : *IntervalProd*, `float` or array-like

The set to be inserted. A `float` or array a is treated as an `IntervalProd(a, a)`.

index : `int`, optional

The index of the dimension before which `other` is to be inserted. Must fulfill `0 <= index <= ndim`.

Default: *ndim*

Returnslarger_set : *IntervalProd*

The enlarged set

Examples

```
>>> rbox = IntervalProd([-1, 2], [-0.5, 3])
>>> rbox2 = IntervalProd([0, 0], [1, 0])
>>> rbox.insert(rbox2, 1)
IntervalProd([-1.0, 0.0, 0.0, 2.0], [-0.5, 1.0, 0.0, 3.0])
>>> rbox.insert([-1.0, 0.0], 2)
IntervalProd([-1.0, 2.0, -1.0, 0.0], [-0.5, 3.0, -1.0, 0.0])
```

Without index, inserts at the end

```
>>> rbox.insert([-1.0, 0.0])
IntervalProd([-1.0, 2.0, -1.0, 0.0], [-0.5, 3.0, -1.0, 0.0])
```

Can also insert by array

```
>>> rbox.insert(0, 1).squeeze() == rbox
True
```

IntervalProd.max

`IntervalProd.`**`max`**`()`

The maximum value in this interval product

IntervalProd.measure

`IntervalProd.`**`measure`**`(`*ndim=None*`)`

The (Lebesgue) measure of this interval product.

Parametersndim : `int`, optional

The dimension of the measure to apply. Default: *true_ndim*

Examples

```
>>> b, e = [-1, 2.5, 0], [-0.5, 10, 0]
>>> rbox = IntervalProd(b, e)
>>> rbox.measure()
3.75
>>> rbox.measure(ndim=3)
0.0
>>> rbox.measure(ndim=3) == rbox.volume
True
>>> rbox.measure(ndim=1)
inf
>>> rbox.measure() == rbox.squeeze().volume
True
```

IntervalProd.min

`IntervalProd.`**`min`**`()`

The minimum value in this interval product

IntervalProd.squeeze

`IntervalProd.`**`squeeze`**`()`

Remove the degenerate dimensions.

Note that no changes are made in-place.

Returnssqueezed : *IntervalProd*

The squeezed set

Examples

```
>>> b, e = [-1, 0, 2], [-0.5, 1, 3]
>>> rbox = IntervalProd(b, e)
>>> rbox.collapse(1, 0).squeeze()
Rectangle([-1.0, 2.0], [-0.5, 3.0])
>>> rbox.collapse([1, 2], [0, 2.5]).squeeze()
Interval(-1.0, -0.5)
```

```
>>> rbox.collapse([0, 1, 2], [-1, 0, 2.5]).squeeze()
IntervalProd([], [])
```

__init__(*begin*, *end*)
Initialize a new instance.

Parametersbegin : array-like or float

The lower ends of the intervals in the product

end : array-like or float

The upper ends of the intervals in the product

Examples

```
>>> b, e = [-1, 2.5, 70, 80], [-0.5, 10, 75, 90]
>>> rbox = IntervalProd(b, e)
>>> rbox
IntervalProd([-1.0, 2.5, 70.0, 80.0], [-0.5, 10.0, 75.0, 90.0])
```

Functions

Cuboid(begin, end)	Three-dimensional interval product.
Interval(begin, end)	One-dimensional interval product.
Rectangle(begin, end)	Two-dimensional interval product.

Cuboid

odl.set.domain.**Cuboid**(*begin*, *end*)
Three-dimensional interval product.

Parametersbegin : array-like with three elements convertible to float

The lower ends of the intervals in the product

end : array-like with three elements convertible to float

The upper ends of the intervals in the product

Interval

odl.set.domain.**Interval**(*begin*, *end*)
One-dimensional interval product.

Parametersbegin : array-like or float

The lower ends of the intervals in the product

end : array-like or float

The upper ends of the intervals in the product

Rectangle

odl.set.domain.**Rectangle**(*begin*, *end*)
Two-dimensional interval product.

Parametersbegin : array-like with two elements convertible to float

The lower ends of the intervals in the product

end : array-like with two elements convertible to float

The upper ends of the intervals in the product

pspace

Cartesian products of *LinearSpace* instances.

Classes

ProductSpace(*spaces, **kwargs)	Cartesian product of *LinearSpace*'s.
ProductSpaceVector(space, parts)	Elements in a *ProductSpace*

ProductSpace

class odl.set.pspace.**ProductSpace**(**spaces*, ***kwargs*)
Bases: *odl.set.space.LinearSpace*

Cartesian product of *LinearSpace*'s.

Attributes

element_type	*ProductSpaceVector*
field	The field of this vector space.
size	The number of factors.
spaces	A tuple containing all spaces.
weights	Weighting vector or scalar of this product space.

ProductSpace.element_type
ProductSpace.**element_type**
ProductSpaceVector

ProductSpace.field
ProductSpace.**field**
The field of this vector space.

The field is the set of scalars of the space, that is numbers that the vectors in the space can be multiplied with.

Returnsfield : *Field*

The underlying field.

ProductSpace.size
`ProductSpace.size`
The number of factors.

ProductSpace.spaces
`ProductSpace.spaces`
A tuple containing all spaces.

ProductSpace.weights
`ProductSpace.weights`
Weighting vector or scalar of this product space.

Methods

__eq__(other)	Return `self == other`.
__getitem__(indices)	Return `self[indices]`.
_dist(x1, x2)	
_inner(x1, x2)	
_lincomb(a, x, b, y, out)	
_multiply(x1, x2, out)	
_norm(x)	
element([inp])	Create an element in the product space.
one()	Create the one vector of the product space.
zero()	Create the zero vector of the product space.

ProductSpace.__eq__
`ProductSpace.__eq__`(*other*)
Return `self == other`.

Returnsequals : `bool`

`True` if `other` is a *ProductSpace* instance, has the same length and the same factors. `False` otherwise.

Examples

```
>>> from odl import Rn
>>> r2, r3 = Rn(2), Rn(3)
>>> rn, rm = Rn(2), Rn(3)
>>> r2x3, rnxm = ProductSpace(r2, r3), ProductSpace(rn, rm)
>>> r2x3 == rnxm
True
>>> r3x2 = ProductSpace(r3, r2)
>>> r2x3 == r3x2
False
>>> r5 = ProductSpace(*[Rn(1)]*5)
>>> r2x3 == r5
False
>>> r5 = Rn(5)
>>> r2x3 == r5
False
```

ProductSpace.__getitem__
`ProductSpace.__getitem__(indices)`
Return `self[indices]`.

ProductSpace._dist
`ProductSpace._dist(x1, x2)`

ProductSpace._inner
`ProductSpace._inner(x1, x2)`

ProductSpace._lincomb
`ProductSpace._lincomb(a, x, b, y, out)`

ProductSpace._multiply
`ProductSpace._multiply(x1, x2, out)`

ProductSpace._norm
`ProductSpace._norm(x)`

ProductSpace.element
`ProductSpace.element(inp=None)`
Create an element in the product space.

Parametersinp : `object`, optional

If `inp` is `None`, a new element is created from scratch by allocation in the spaces. If `inp` is already an element in this space, it is re-wrapped. Otherwise, a new element is created from the components by calling the `element()` methods in the component spaces.

Returnselement : *`ProductSpaceVector`*

The new element

Examples

```
>>> from odl import Rn
>>> r2, r3 = Rn(2), Rn(3)
>>> vec_2, vec_3 = r2.element(), r3.element()
>>> r2x3 = ProductSpace(r2, r3)
>>> vec_2x3 = r2x3.element()
>>> vec_2.space == vec_2x3[0].space
True
>>> vec_3.space == vec_2x3[1].space
True
```

Creates an element in the product space

```
>>> from odl import Rn
>>> r2, r3 = Rn(2), Rn(3)
>>> prod = ProductSpace(r2, r3)
>>> x2 = r2.element([1, 2])
```

```
>>> x3 = r3.element([1, 2, 3])
>>> x = prod.element([x2, x3])
>>> print(x)
{[1.0, 2.0], [1.0, 2.0, 3.0]}
```

ProductSpace.one

`ProductSpace.one()`

Create the one vector of the product space.

The i:th component of the product space one vector is the one vector of the i:th space in the product.

ParametersNone

Returnsone : ProductSpaceVector

The one vector in the product space

Examples

```
>>> from odl import Rn
>>> r2, r3 = Rn(2), Rn(3)
>>> one_2, one_3 = r2.one(), r3.one()
>>> r2x3 = ProductSpace(r2, r3)
>>> one_2x3 = r2x3.one()
>>> one_2 == one_2x3[0]
True
>>> one_3 == one_2x3[1]
True
```

ProductSpace.zero

`ProductSpace.zero()`

Create the zero vector of the product space.

The i:th component of the product space zero vector is the zero vector of the i:th space in the product.

ParametersNone

Returnszero : ProductSpaceVector

The zero vector in the product space

Examples

```
>>> from odl import Rn
>>> r2, r3 = Rn(2), Rn(3)
>>> zero_2, zero_3 = r2.zero(), r3.zero()
>>> r2x3 = ProductSpace(r2, r3)
>>> zero_2x3 = r2x3.zero()
>>> zero_2 == zero_2x3[0]
True
>>> zero_3 == zero_2x3[1]
True
```

__init__(**spaces*, ***kwargs*)

Initialize a new instance.

The Cartesian product $\mathcal{X}_1 \times \cdots \times \mathcal{X}_n$ for linear spaces $\mathcal{X}_i$ is itself a linear space, where the linear combination is defined component-wise.

Parametersspaces : *LinearSpace* or `int`

Can be specified either as a space and an integer, in which case the power space `space**n` is created, or an arbitrary number of spaces.

kwargs : {'ord', 'weights', 'prod_norm'}

'ord'[`float`, optional] Order of the product distance/norm, i.e.

```
dist(x, y) = np.linalg.norm(x-y, ord=ord)
```

```
norm(x) = np.linalg.norm(x, ord=ord)
```

Default: 2.0

The following `float` values for `ord` can be specified. Note that any value of `ord < 1` only gives a pseudo-norm.

'prod_norm'	Distance Definition
'inf'	`max(w * z)`
'-inf'	`min(w * z)`
other	`sum(w * z**ord)**(1/ord)`

Here,

```
z = (x[0].dist(y[0]),..., x[n-1].dist(y[n-1]))
```

and `w = weights`.

Note that `0 <= ord < 1` are not allowed since these pseudo-norms are very unstable numerically.

'weights'[array-like, optional, only usable with 'ord'] Array of weights, same size as number of space components. All weights must be positive. It is multiplied with the tuple of distances before applying the Rn norm or `prod_norm`.

Default: `(1.0,...,1.0)`

'prod_norm'[`callable`, optional] Function that should be applied to the array of distances/norms. Specifying a product norm causes the space to NOT be a Hilbert space.

Default: `np.linalg.norm(x, ord=ord)`.

Returnsprodspace : *ProductSpace*

Examples

```
>>> from odl import Rn
>>> r2x3 = ProductSpace(Rn(2), Rn(3))
```

ProductSpaceVector

class `odl.set.pspace.`**`ProductSpaceVector`**(*space*, *parts*)

Bases: *odl.set.space.LinearSpaceVector*

Elements in a *ProductSpace*

Attributes

T	The transpose of a vector, the functional given by (.
parts	The parts of this vector.
size	The number of factors of this vector's space.
space	Space to which this vector belongs.
ufunc	*ProductSpaceUFuncs*, access to numpy style ufuncs.

ProductSpaceVector.T

`ProductSpaceVector.`**`T`**

The transpose of a vector, the functional given by (. , self)

Returnstranspose : *InnerProductOperator*

Notes

This function is only defined in inner product spaces.

In a complex space, this takes the conjugate transpose of the vector.

Examples

```
>>> from odl import Rn
>>> import numpy as np
>>> rn = Rn(3)
>>> x = rn.element([1, 2, 3])
>>> y = rn.element([2, 1, 3])
>>> x.T(y)
13.0
```

ProductSpaceVector.parts

`ProductSpaceVector.`**`parts`**

The parts of this vector.

ProductSpaceVector.size

`ProductSpaceVector.`**`size`**

The number of factors of this vector's space.

ProductSpaceVector.space

`ProductSpaceVector.`**`space`**

Space to which this vector belongs.

LinearSpace

ProductSpaceVector.ufunc

`ProductSpaceVector.`**`ufunc`**

ProductSpaceUFuncs, access to numpy style ufuncs.

These are always available if the underlying spaces are *NtuplesBase*.

See also:

NtuplesBaseUFuncsBase class for ufuncs in *NtuplesBase* spaces, sub spaces may override this for greater efficiency.

ProductSpaceUFuncsFor a list of available ufuncs.

Examples

```
>>> from odl import Rn
>>> r22 = ProductSpace(Rn(2), 2)
>>> x = r22.element([[1, -2], [-3, 4]])
>>> x.ufunc.absolute()
ProductSpace(Rn(2), 2).element([
    [1.0, 2.0],
    [3.0, 4.0]
])
```

These functions can also be used with non-vector arguments and support broadcasting, both by element

```
>>> x.ufunc.add([1, 1])
ProductSpace(Rn(2), 2).element([
    [2.0, -1.0],
    [-2.0, 5.0]
])
```

and also recursively

```
>>> x.ufunc.subtract(1)
ProductSpace(Rn(2), 2).element([
    [0.0, -3.0],
    [-4.0, 3.0]
])
```

There is also support for various reductions (sum, prod, min, max)

```
>>> x.ufunc.sum()
0.0
```

Also supports out parameter

```
>>> y = r22.element()
>>> result = x.ufunc.absolute(out=y)
>>> result
ProductSpace(Rn(2), 2).element([
    [1.0, 2.0],
    [3.0, 4.0]
])
>>> result is y
True
```

Methods

__eq__(other)	Return `self == other`.
__getitem__(indices)	Return `self[indices]`.
__setitem__(indices, values)	Implement `self[indices] = vals`.
show(*args, **kwargs)	Display the parts of this vector.

ProductSpaceVector.__eq__
`ProductSpaceVector.__eq__`(*other*)
Return `self == other`.

Overrides the default *LinearSpace* method since it is implemented with the distance function, which is prone to numerical errors. This function checks equality per component.

ProductSpaceVector.__getitem__
`ProductSpaceVector.__getitem__`(*indices*)
Return `self[indices]`.

ProductSpaceVector.__setitem__
`ProductSpaceVector.__setitem__`(*indices*, *values*)
Implement `self[indices] = vals`.

ProductSpaceVector.show
`ProductSpaceVector.show`(**args*, ***kwargs*)
Display the parts of this vector.
`__init__`(*space*, *parts*)
"Initialize a new instance.

sets

Basic abstract and concrete sets.

Classes

CartesianProduct(*sets)	The Cartesian product of `n` sets.
ComplexNumbers	The set of complex numbers.
EmptySet	The empty set.
Field	Any set that satisfies the field axioms
Integers	The set of integers.
RealNumbers	The set of real numbers.
Set	An abstract set.
Strings(length)	The set of fixed-length (unicode) strings.
UniversalSet	The set of all objects.

CartesianProduct

class `odl.set.sets.CartesianProduct`(**sets*)
Bases: *odl.set.sets.Set*

The Cartesian product of `n` sets.

The elements of this set are `n`-tuples where the i-th entry is an element of the i-th set.

Attributes

sets	The factors (sets) as a `tuple`.

CartesianProduct.sets

`CartesianProduct.`**`sets`**

The factors (sets) as a `tuple`.

Methods

__contains__(other)	Test if `other` is contained in this set.
__eq__(other)	Return `self == other`.
__getitem__(indcs)	Return `self[indcs]`.
contains_set(other)	Test if `other` is a subset of this set.
element([inp])	Create a *CartesianProduct* element.

CartesianProduct.__contains__

`CartesianProduct.`**`__contains__`**(*other*)

Test if `other` is contained in this set.

Returnscontains : `bool`

`True` if `other` has the same length as this Cartesian product and each entry is contained in the set with corresponding index, `False` otherwise.

CartesianProduct.__eq__

`CartesianProduct.`**`__eq__`**(*other*)

Return `self == other`.

Returnsequals : `bool`

`True` if `other` is a *CartesianProduct* instance, has the same length as this Cartesian product and all sets with the same index are equal, `False` otherwise.

CartesianProduct.__getitem__

`CartesianProduct.`**`__getitem__`**(*indcs*)

Return `self[indcs]`.

Examples

```
>>> emp, univ = EmptySet(), UniversalSet()
>>> prod = CartesianProduct(emp, univ, univ, emp, emp)
>>> prod[2]
UniversalSet()
>>> prod[2:4]
CartesianProduct(UniversalSet(), EmptySet())
```

CartesianProduct.contains_set

`CartesianProduct.`**`contains_set`**(*other*)

Test if `other` is a subset of this set.

Implementing this method is optional. Default it tests for equality.

CartesianProduct.element
`CartesianProduct.`**`element`**(*inp=None*)
Create a *CartesianProduct* element.

Parametersinp : `iterable`, optional

Collection of input values for the *LinearSpace.element* methods of all sets in the Cartesian product.

Returnselement : `tuple`

A tuple of the given input

`__init__`(**sets*)
Initialize a new instance.

ComplexNumbers

class `odl.set.sets.`**`ComplexNumbers`**
Bases: *odl.set.sets.Field*

The set of complex numbers.

Attributes

field	The field of scalars for a field is itself.

ComplexNumbers.field
`ComplexNumbers.`**`field`**
The field of scalars for a field is itself.

Notes

This is a hack for this to work with duck-typing with *LinearSpace*'s.

Methods

__contains__(other)	Test if `other` is a complex number.
__eq__(other)	Return `self == other`.
contains_set(other)	Test if `other` is a subset of the complex numbers
element([inp])	Return a complex number from `inp` or from scratch.

ComplexNumbers.__contains__
`ComplexNumbers.`**`__contains__`**(*other*)
Test if `other` is a complex number.

ComplexNumbers.__eq__
`ComplexNumbers.`**`__eq__`**(*other*)
Return `self == other`.

ComplexNumbers.contains_set
ComplexNumbers.**contains_set**(*other*)
Test if other is a subset of the complex numbers

Returnscontained : bool

True if other is *ComplexNumbers*, *RealNumbers* or *Integers*, False else.

Examples

```
>>> C = ComplexNumbers()
>>> C.contains_set(RealNumbers())
True
```

ComplexNumbers.element
ComplexNumbers.**element**(*inp=None*)
Return a complex number from inp or from scratch.

EmptySet

class odl.set.sets.**EmptySet**
Bases: *odl.set.sets.Set*

The empty set.

None is considered as "no element", i.e. None in EmptySet() is True

Methods

__contains__(other)	Test if other is None.
__eq__(other)	Return self == other.
contains_set(other)	Return True for the empty set, otherwise False.
element([inp])	Return None.

EmptySet.__contains__
EmptySet.**__contains__**(*other*)
Test if other is None.

EmptySet.__eq__
EmptySet.**__eq__**(*other*)
Return self == other.

EmptySet.contains_set
EmptySet.**contains_set**(*other*)
Return True for the empty set, otherwise False.

EmptySet.element
EmptySet.**element**(*inp=None*)
Return None.

Field

class odl.set.sets.**Field**

Bases: *odl.set.sets.Set*

Any set that satisfies the field axioms

For example *RealNumbers*, *ComplexNumbers* or the finite field F_2.

Attributes

field	The field of scalars for a field is itself.

Field.field

Field.**field**

The field of scalars for a field is itself.

Notes

This is a hack for this to work with duck-typing with *LinearSpace*'s.

Methods

__contains__(other)	Return `other in self`.
__eq__(other)	Return `self == other`.
contains_set(other)	Test if `other` is a subset of this set.
element([inp])	Return an element from `inp` or from scratch.

Field.__contains__

Field.**__contains__**(*other*)

Return `other in self`.

Field.__eq__

Field.**__eq__**(*other*)

Return `self == other`.

Field.contains_set

Field.**contains_set**(*other*)

Test if `other` is a subset of this set.

Implementing this method is optional. Default it tests for equality.

Field.element

Field.**element**(*inp=None*)

Return an element from `inp` or from scratch.

Implementing this method is optional.

Integers

class odl.set.sets.**Integers**
Bases: *odl.set.sets.Set*

The set of integers.

Methods

__contains__(other)	Test if `other` is an integer.
__eq__(other)	Return `self == other`.
contains_set(other)	Test if `other` is a subset of the real numbers
element([inp])	Return an integer from `inp` or from scratch.

Integers.__contains__
Integers.**__contains__**(*other*)
Test if `other` is an integer.

Integers.__eq__
Integers.**__eq__**(*other*)
Return `self == other`.

Integers.contains_set
Integers.**contains_set**(*other*)
Test if `other` is a subset of the real numbers

Returnscontained : `bool`

`True` if other is *Integers*, `False` otherwise.

Examples

```
>>> Z = Integers()
>>> Z.contains_set(RealNumbers())
False
```

Integers.element
Integers.**element**(*inp=None*)
Return an integer from `inp` or from scratch.

RealNumbers

class odl.set.sets.**RealNumbers**
Bases: *odl.set.sets.Field*

The set of real numbers.

Attributes

field	The field of scalars for a field is itself.

RealNumbers.field
RealNumbers.**field**
The field of scalars for a field is itself.

Notes

This is a hack for this to work with duck-typing with *LinearSpace*'s.

Methods

__contains__(other)	Test if other is a real number.
__eq__(other)	Return self == other.
contains_set(other)	Test if other is a subset of the real numbers
element([inp])	Return a real number from inp or from scratch.

RealNumbers.__contains__
RealNumbers.**__contains__**(*other*)
Test if other is a real number.

RealNumbers.__eq__
RealNumbers.**__eq__**(*other*)
Return self == other.

RealNumbers.contains_set
RealNumbers.**contains_set**(*other*)
Test if other is a subset of the real numbers

Returnscontained : bool

True if other is *RealNumbers* or *Integers* False else.

Examples

```
>>> R = RealNumbers()
>>> R.contains_set(RealNumbers())
True
```

RealNumbers.element
RealNumbers.**element**(*inp=None*)
Return a real number from inp or from scratch.

Set

class odl.set.sets.**Set**
Bases: object

An abstract set.

Abstract Methods

Each subclass of *Set* must implement two methods: one to check if an object is contained in the set and one to test if two sets are equal.

Membership test: `__contains__(self, other)`

Test if `other` is a member of this set. This function provides the operator overload for `in`.

Parameters:

other[`object`] The object to be tested for membership

Returns:

contains[`bool`] `True` if `other` is a member of this set, `False` otherwise.

Equality test: `__eq__(self, other)`

Test if `other` is the same set as this set, i.e. both sets are of the same type and contain the same elements. This function provides the operator overload for ==.

Parameters:

other[`object`] The object to be tested for equality.

Returns:

equals[`bool`] `True` if both sets are of the same type and contain the same elements, `False` otherwise.

A default implementation of the operator overload for `!=` via `__ne__(self, other)` is provided as `not self.__eq__(other)`.

Element creation (optional): `element(self, inp=None)`

Create an element of this set, either from scratch or from an input parameter.

Parameters:

inp[`object`, optional] The object from which to create the new element

Returns:

element[member of this set] If `inp` is `None`, return an arbitrary element. Otherwise, return the element created from `inp`.

Methods

__contains__(other)	Return `other in self`.
__eq__(other)	Return `self == other`.
contains_set(other)	Test if `other` is a subset of this set.
element([inp])	Return an element from `inp` or from scratch.

Set.__contains__

`Set.__contains__`(*other*)

Return `other in self`.

Set.__eq__

`Set.__eq__`(*other*)

Return `self == other`.

Set.contains_set
`Set.`**`contains_set`**`(`*other*`)`
Test if `other` is a subset of this set.

Implementing this method is optional. Default it tests for equality.

Set.element
`Set.`**`element`**`(`*inp=None*`)`
Return an element from `inp` or from scratch.

Implementing this method is optional.

Strings

class `odl.set.sets.`**`Strings`**`(`*length*`)`
Bases: *odl.set.sets.Set*

The set of fixed-length (unicode) strings.

Attributes

length	The length attribute.

Strings.length
`Strings.`**`length`**
The length attribute.

Methods

__contains__(other)	Return `other in self`.
__eq__(other)	Return `self == other`.
contains_set(other)	Test if `other` is a subset of this set.
element([inp])	Return a string from `inp` or from scratch.

Strings.__contains__
`Strings.`**`__contains__`**`(`*other*`)`
Return `other in self`.

`True` if `other` is a string of at max *length* characters, `False` otherwise.

Strings.__eq__
`Strings.`**`__eq__`**`(`*other*`)`
Return `self == other`.

Strings.contains_set
`Strings.`**`contains_set`**`(`*other*`)`
Test if `other` is a subset of this set.

Implementing this method is optional. Default it tests for equality.

Strings.element
`Strings.element(inp=None)`
Return a string from `inp` or from scratch.
__init__(*length*)
Initialize a new instance.

Parameterslength : `int`

The fixed length of the strings in this set. Must be positive.

UniversalSet

class `odl.set.sets.UniversalSet`
Bases: *odl.set.sets.Set*

The set of all objects.

Forget about set theory for a moment :-).

Methods

__contains__(other)	Return `True`.
__eq__(other)	Return `self == other`.
contains_set(other)	Return `True` for any set.
element([inp])	Return `inp` in any case.

UniversalSet.__contains__
`UniversalSet.__contains__(other)`
Return `True`.

UniversalSet.__eq__
`UniversalSet.__eq__(other)`
Return `self == other`.

UniversalSet.contains_set
`UniversalSet.contains_set(other)`
Return `True` for any set.

UniversalSet.element
`UniversalSet.element(inp=None)`
Return `inp` in any case.

space

Abstract linear vector spaces.

The classes in this module represent abstract mathematical concepts of vector spaces. They cannot be used directly but are rather intended to be sub-classed by concrete space implementations. The spaces provide default implementations of the most important vector space operations. See the documentation of the respective classes for more details.

The abstract *LinearSpace* class is intended for quick prototyping. It has a number of abstract methods which must be overridden by a subclass. On the other hand, it provides automatic error checking and numerous attributes and methods for convenience.

Classes

LinearSpace(field)	Abstract linear vector space.
LinearSpaceNotImplementedError	Exception for not implemented errors in *LinearSpace*'s.
LinearSpaceTypeError	Exception for type errors in *LinearSpace*'s.
LinearSpaceVector(space)	Abstract *LinearSpace* element.
UniversalSpace()	A dummy linear space class.

LinearSpace

class odl.set.space.**LinearSpace**(*field*)

Bases: *odl.set.sets.Set*

Abstract linear vector space.

Its elements are represented as instances of the inner *LinearSpaceVector* class.

Attributes

element_type	*LinearSpaceVector*
field	The field of this vector space.

LinearSpace.element_type

LinearSpace.**element_type**

LinearSpaceVector

LinearSpace.field

LinearSpace.**field**

The field of this vector space.

The field is the set of scalars of the space, that is numbers that the vectors in the space can be multiplied with.

Returnsfield : *Field*

The underlying field.

Methods

__contains__(other)	Return `other in self`.
_dist(x1, x2)	Calculate the distance between x1 and x2.
_inner(x1, x2)	Calculate the inner product of x1 and x2.
_lincomb(a, x1, b, x2, out)	Calculate `out = a*x1 + b*x2`.
_multiply(x1, x2, out)	Calculate the pointwise multiplication out = x1 * x2.

Continued on next page

Table 2.122 – continued from previous page

_norm(x)	Calculate the norm of x.
dist(x1, x2)	Calculate the distance between two vectors.
divide(x1, x2[, out])	Calculate the pointwise division of `x1` and `x2`
element([inp])	Create a *LinearSpaceVector* from `inp` or from scratch.
inner(x1, x2)	Calculate the inner product of `x1` and `x2`.
lincomb(a, x1[, b, x2, out])	Linear combination of vectors.
multiply(x1, x2[, out])	Calculate the pointwise product of `x1` and `x2`.
norm(x)	Calculate the norm of a vector.
one()	A one vector in this space.
zero()	A zero vector in this space.

LinearSpace.__contains__

`LinearSpace.__contains__`(*other*)

Return `other in self`.

Returnscontains : `bool`

`True` if `other` is a *LinearSpaceVector* instance and `other.space` is equal to this space, `False` otherwise.

Notes

This is the strict default where spaces must be equal. Subclasses may choose to implement a less strict check.

LinearSpace._dist

`LinearSpace._dist`(*x1*, *x2*)

Calculate the distance between x1 and x2.

This method is intended to be private, public callers should resort to *dist* which is type-checked.

LinearSpace._inner

`LinearSpace._inner`(*x1*, *x2*)

Calculate the inner product of x1 and x2.

This method is intended to be private, public callers should resort to *inner* which is type-checked.

LinearSpace._lincomb

`LinearSpace._lincomb`(*a*, *x1*, *b*, *x2*, *out*)

Calculate `out = a*x1 + b*x2`.

This method is intended to be private, public callers should resort to *lincomb* which is type-checked.

LinearSpace._multiply

`LinearSpace._multiply`(*x1*, *x2*, *out*)

Calculate the pointwise multiplication out = x1 * x2.

This method is intended to be private, public callers should resort to *multiply* which is type-checked.

LinearSpace._norm
`LinearSpace._norm(x)`
Calculate the norm of x.

This method is intended to be private, public callers should resort to *norm* which is type-checked.

LinearSpace.dist
`LinearSpace.dist(x1, x2)`
Calculate the distance between two vectors.

Parametersx1 : *LinearSpaceVector*

The first element

x2 : *LinearSpaceVector*

The second element

Returnsdist : `float`

Distance between vectors

LinearSpace.divide
`LinearSpace.divide(x1, x2, out=None)`
Calculate the pointwise division of `x1` and `x2`

Parametersx1 : *LinearSpaceVector*

The dividend

x2 : *LinearSpaceVector*

The divisor

out : *LinearSpaceVector*, optional

Vector to write the ratio to. default: *LinearSpace.element*

Returnsout : *LinearSpaceVector*

Ratio of the vectors, same as `out` if given.

LinearSpace.element
`LinearSpace.element(inp=None, **kwargs)`
Create a *LinearSpaceVector* from `inp` or from scratch.

If called without `inp` argument, an arbitrary element in the space is generated without guarantee of its state.

Parametersinp : `object`, optional

The input data from which to create the element

****args** : `dict`, optional

Optional further arguments.

Returnselement : *LinearSpaceVector*

A vector in this space

LinearSpace.inner
`LinearSpace.`**`inner`**(*x1*, *x2*)

Calculate the inner product of `x1` and `x2`.

Parametersx1 : *LinearSpaceVector*

The first vector

x2 : *LinearSpaceVector*

The second vector

Returnsout : *LinearSpace.field* element

Product of the vectors, same as `out` if given.

LinearSpace.lincomb
`LinearSpace.`**`lincomb`**(*a*, *x1*, *b=None*, *x2=None*, *out=None*)

Linear combination of vectors.

Calculates

```
out = a * x1
```

or, if b and y are given,

```
out = a*x1 + b*x2
```

with error checking of types.

Parametersa : Scalar in the field of this space

Scalar to multiply `x1` with.

x1 : *LinearSpaceVector*

The first of the summands

b : Scalar, optional

Scalar to multiply `x2` with.

x2 : *LinearSpaceVector*, optional

The second of the summands

out : *LinearSpaceVector*, optional

The Vector that the result should be written to.

Returnsout : *LinearSpaceVector*

Notes

The vectors `out`, `x1` and `x2` may be aligned, thus a call

```
space.lincomb(x, 2, x, 3.14, out=x)
```

is (mathematically) equivalent to

```
x = x * (1 + 2 + 3.14)
```

LinearSpace.multiply
`LinearSpace.`**`multiply`**(*x1*, *x2*, *out=None*)
Calculate the pointwise product of `x1` and `x2`.

Parametersx1 : *LinearSpaceVector*

The first multiplicand

x2 : *LinearSpaceVector*

The second multiplicand

out : *LinearSpaceVector*, optional

Vector to write the product to. default: *LinearSpace.element*

Returnsout : *LinearSpaceVector*

Product of the vectors, same as `out` if given.

LinearSpace.norm
`LinearSpace.`**`norm`**(*x*)
Calculate the norm of a vector.

Parametersx : *LinearSpaceVector*

The vector

Returnsout : `float`

Norm of the vector

LinearSpace.one
`LinearSpace.`**`one`**()
A one vector in this space.

The one vector is defined as the multiplicative unit of a space.

Returnsv : *LinearSpaceVector*

The one vector of this space

LinearSpace.zero
`LinearSpace.`**`zero`**()
A zero vector in this space.

The zero vector is defined as the additive unit of a space.

Returnsv : *LinearSpaceVector*

The zero vector of this space

`__init__`(*field*)
Initialize a LinearSpace.

This method should be called by all inheriting methods so that the field property of the space is set properly.

Parametersfield : *Field*

The underlying scalar field of the space

LinearSpaceNotImplementedError

exception odl.set.space.**LinearSpaceNotImplementedError**
Exception for not implemented errors in *LinearSpace*'s.

These are raised when a method in *LinearSpace* that has not been defined in a specific space is called.

LinearSpaceTypeError

exception odl.set.space.**LinearSpaceTypeError**
Exception for type errors in *LinearSpace*'s.

These are raised when the wrong type of element is fed to *LinearSpace.lincomb* and related functions.

LinearSpaceVector

class odl.set.space.**LinearSpaceVector**(*space*)
Bases: object

Abstract *LinearSpace* element.

Not intended for creation of vectors, use the space's *LinearSpace.element* method instead.

Attributes

T	The transpose of a vector, the functional given by (.
space	Space to which this vector belongs.

LinearSpaceVector.T
LinearSpaceVector.**T**
The transpose of a vector, the functional given by (. , self)

Returnstranspose : *InnerProductOperator*

Notes

This function is only defined in inner product spaces.

In a complex space, this takes the conjugate transpose of the vector.

Examples

```
>>> from odl import Rn
>>> import numpy as np
>>> rn = Rn(3)
>>> x = rn.element([1, 2, 3])
>>> y = rn.element([2, 1, 3])
>>> x.T(y)
13.0
```

LinearSpaceVector.space
`LinearSpaceVector.`**`space`**
Space to which this vector belongs.

LinearSpace

Methods

__eq__(other)	Return `self == other`.
assign(other)	Assign the values of `other` to self.
copy()	Create an identical (deep) copy of self.
dist(other)	Distance to `other`.
inner(other)	Inner product with `other`.
lincomb(a, x1[, b, x2])	Assign a linear combination to this vector.
multiply(x, y)	Multiply by `other` inplace.
norm()	Norm of vector
set_zero()	Set this vector to zero.

LinearSpaceVector.__eq__
`LinearSpaceVector.`**`__eq__`**(*other*)
Return `self == other`.

Two vectors are equal if their distance is 0

Parametersother : *LinearSpaceVector*

Vector in this space.

Returnsequals : `bool`

True if the vectors are equal, else false.

Notes

Equality is very sensitive to numerical errors, thus any operations on a vector should be expected to break equality testing.

Examples

```
>>> from odl import Rn
>>> import numpy as np
>>> rn = Rn(1, norm=np.linalg.norm)
>>> x = rn.element([0.1])
>>> x == x
True
>>> y = rn.element([0.1])
>>> x == y
True
>>> z = rn.element([0.3])
>>> x+x+x == z
False
```

LinearSpaceVector.assign
`LinearSpaceVector.`**`assign`**(*other*)
Assign the values of `other` to self.

LinearSpaceVector.copy
`LinearSpaceVector.`**`copy`**()
Create an identical (deep) copy of self.

LinearSpaceVector.dist
`LinearSpaceVector.`**`dist`**(*other*)
Distance to `other`.

LinearSpace.dist

LinearSpaceVector.inner
`LinearSpaceVector.`**`inner`**(*other*)
Inner product with `other`.

LinearSpace.inner

LinearSpaceVector.lincomb
`LinearSpaceVector.`**`lincomb`**(*a*, *x1*, *b=None*, *x2=None*)
Assign a linear combination to this vector.

Implemented as `space.lincomb(a, x1, b, x2, out=self)`.

LinearSpace.lincomb

LinearSpaceVector.multiply
`LinearSpaceVector.`**`multiply`**(*x*, *y*)
Multiply by `other` inplace.

LinearSpace.multiply

LinearSpaceVector.norm
`LinearSpaceVector.`**`norm`**()
Norm of vector

LinearSpace.norm

LinearSpaceVector.set_zero
`LinearSpaceVector.`**`set_zero`**()
Set this vector to zero.

LinearSpace.zero
`__init__`(*space*)
Default initializer of vectors.

All deriving classes must call this method to set space.

UniversalSpace

class odl.set.space.**UniversalSpace**
Bases: *odl.set.space.LinearSpace*

A dummy linear space class.

Mostly raising *LinearSpaceNotImplementedError*.

Attributes

element_type	*LinearSpaceVector*
field	The field of this vector space.

UniversalSpace.element_type
UniversalSpace.**element_type**
LinearSpaceVector

UniversalSpace.field
UniversalSpace.**field**
The field of this vector space.

The field is the set of scalars of the space, that is numbers that the vectors in the space can be multiplied with.

Returnsfield : *Field*

The underlying field.

Methods

__contains__(other)	Return `other in self`.
__eq__(other)	Return `self == other`.
_dist(x1, x2)	Dummy distance method.
_divide(x1, x2, out)	Dummy division method.
_inner(x1, x2)	Dummy inner product method.
_lincomb(a, x1, b, x2, out)	Dummy linear combination.
_multiply(x1, x2, out)	Dummy multiplication method.
_norm(x)	Dummy norm method.
dist(x1, x2)	Calculate the distance between two vectors.
divide(x1, x2[, out])	Calculate the pointwise division of `x1` and `x2`
element([inp])	Dummy element creation method.
inner(x1, x2)	Calculate the inner product of `x1` and `x2`.
lincomb(a, x1[, b, x2, out])	Linear combination of vectors.
multiply(x1, x2[, out])	Calculate the pointwise product of `x1` and `x2`.
norm(x)	Calculate the norm of a vector.
one()	A one vector in this space.
zero()	A zero vector in this space.

UniversalSpace.__contains__

UniversalSpace.**__contains__**(*other*)
Return `other in self`.

Dummy membership check, `True` for any *LinearSpaceVector*.

UniversalSpace.__eq__
UniversalSpace.**__eq__**(*other*)
Return `self == other`.

Dummy check, `True` for any *LinearSpace*.

UniversalSpace._dist
UniversalSpace.**_dist**(*x1*, *x2*)
Dummy distance method.

raises *LinearSpaceNotImplementedError*.

UniversalSpace._divide
UniversalSpace.**_divide**(*x1*, *x2*, *out*)
Dummy division method.

raises *LinearSpaceNotImplementedError*.

UniversalSpace._inner
UniversalSpace.**_inner**(*x1*, *x2*)
Dummy inner product method.

raises *LinearSpaceNotImplementedError*.

UniversalSpace._lincomb
UniversalSpace.**_lincomb**(*a*, *x1*, *b*, *x2*, *out*)
Dummy linear combination.

raises *LinearSpaceNotImplementedError*.

UniversalSpace._multiply
UniversalSpace.**_multiply**(*x1*, *x2*, *out*)
Dummy multiplication method.

raises *LinearSpaceNotImplementedError*.

UniversalSpace._norm
UniversalSpace.**_norm**(*x*)
Dummy norm method.

raises *LinearSpaceNotImplementedError*.

UniversalSpace.dist
UniversalSpace.**dist**(*x1*, *x2*)
Calculate the distance between two vectors.

Parametersx1 : *LinearSpaceVector*

The first element

x2 : *LinearSpaceVector*

The second element

Returnsdist : `float`

Distance between vectors

UniversalSpace.divide
`UniversalSpace.`**`divide`**(*x1*, *x2*, *out=None*)

Calculate the pointwise division of `x1` and `x2`

Parametersx1 : *LinearSpaceVector*

The dividend

x2 : *LinearSpaceVector*

The divisor

out : *LinearSpaceVector*, optional

Vector to write the ratio to. default: *LinearSpace.element*

Returnsout : *LinearSpaceVector*

Ratio of the vectors, same as `out` if given.

UniversalSpace.element
`UniversalSpace.`**`element`**(*inp=None*)

Dummy element creation method.

raises *LinearSpaceNotImplementedError*.

UniversalSpace.inner
`UniversalSpace.`**`inner`**(*x1*, *x2*)

Calculate the inner product of `x1` and `x2`.

Parametersx1 : *LinearSpaceVector*

The first vector

x2 : *LinearSpaceVector*

The second vector

Returnsout : *LinearSpace.field* element

Product of the vectors, same as `out` if given.

UniversalSpace.lincomb
`UniversalSpace.`**`lincomb`**(*a*, *x1*, *b=None*, *x2=None*, *out=None*)

Linear combination of vectors.

Calculates

```
out = a * x1
```

or, if b and y are given,

```
out = a*x1 + b*x2
```

with error checking of types.

Parametersa : Scalar in the field of this space

Scalar to multiply `x1` with.

x1 : *LinearSpaceVector*

The first of the summands

b : Scalar, optional

Scalar to multiply `x2` with.

x2 : *LinearSpaceVector*, optional

The second of the summands

out : *LinearSpaceVector*, optional

The Vector that the result should be written to.

Returnsout : *LinearSpaceVector*

Notes

The vectors `out`, `x1` and `x2` may be aligned, thus a call

```
space.lincomb(x, 2, x, 3.14, out=x)
```

is (mathematically) equivalent to

```
x = x * (1 + 2 + 3.14)
```

UniversalSpace.multiply

`UniversalSpace.`**`multiply`**(*x1*, *x2*, *out=None*)

Calculate the pointwise product of `x1` and `x2`.

Parametersx1 : *LinearSpaceVector*

The first multiplicand

x2 : *LinearSpaceVector*

The second multiplicand

out : *LinearSpaceVector*, optional

Vector to write the product to. default: *LinearSpace.element*

Returnsout : *LinearSpaceVector*

Product of the vectors, same as `out` if given.

UniversalSpace.norm

`UniversalSpace.`**`norm`**(*x*)

Calculate the norm of a vector.

Parametersx : *LinearSpaceVector*

The vector

Returnsout : `float`

Norm of the vector

UniversalSpace.one
`UniversalSpace.one()`
A one vector in this space.

The one vector is defined as the multiplicative unit of a space.

Returnsv : *LinearSpaceVector*

The one vector of this space

UniversalSpace.zero
`UniversalSpace.zero()`
A zero vector in this space.

The zero vector is defined as the additive unit of a space.

Returnsv : *LinearSpaceVector*

The zero vector of this space
`__init__()`
Initialize a universal space

2.4.5 solvers

Modules

findroot

Modules

newton

(Quasi-)Newton schemes to find zeros of functions (gradients).

Functions

bfgs_method(grad, x, line_search[, niter, ...])	Quasi-Newton BFGS method to minimize a differentiable function.
broydens_first_method(grad, x, line_search)	Broyden's first method, a quasi-Newton scheme.
broydens_second_method(grad, x, line_search)	Broyden's first method, a quasi-Newton scheme.

bfgs_method
`odl.solvers.findroot.newton.bfgs_method(grad, x, line_search, niter=1, partial=None)`
Quasi-Newton BFGS method to minimize a differentiable function.

This is a general and optimized implementation of a quasi-Newton method with BFGS update for solving a general unconstrained optimization problem

$$\min f(x)$$

for a differentiable function $f : \mathcal{X} \to \mathbb{R}$ on a Hilbert space $\mathcal{X}$. It does so by finding a zero of the gradient

$$\nabla f : \mathcal{X} \to \mathcal{X}.$$

The QN method is an approximate Newton method, where the Hessian is approximated and gradually updated

in each step. This implementation uses the rank-one BFGS update schema where the inverse of the Hessian is recalculated in each iteration.

The algorithm is described in *[R1]*, Section 12.3 and in the BFGS Wikipedia article

Parametersgrad : *Operator*

Gradient mapping of the objective function, i.e. the mapping $x \mapsto \nabla f(x) \in \mathcal{X}$

x : element in the domain of `grad`

Starting point of the iteration

line_search : *LineSearch*

Strategy to choose the step length

niter : `int`, optional

Number of iterations

partial : *Partial*, optional

Object executing code per iteration, e.g. plotting each iterate

Returns`None`

References

[R1]

broydens_first_method

`odl.solvers.findroot.newton.`**`broydens_first_method`**(*grad*, *x*, *line_search*, *niter=1*, *partial=None*)

Broyden's first method, a quasi-Newton scheme.

This is a general and optimized implementation of Broyden's first (or 'good') method, a quasi-Newton method for solving a general unconstrained optimization problem

$$\min f(x)$$

for a differentiable function $f : \mathcal{X} \to \mathbb{R}$ on a Hilbert space $\mathcal{X}$. It does so by finding a zero of the gradient

$$\nabla f : \mathcal{X} \to \mathcal{X}$$

using a Newton-type update scheme with approximate Hessian.

The algorithm is described in *[R2]* and *[R3]*, and in a Wikipedia article.

Parametersgrad : *Operator*

Gradient mapping of the objective function, i.e. the mapping $x \mapsto \nabla f(x) \in \mathcal{X}$

x : element in the domain of `grad`

Starting point of the iteration

line_search : *LineSearch*

Strategy to choose the step length

niter : `int`, optional

Number of iterations

partial : *Partial*, optional

Object executing code per iteration, e.g. plotting each iterate

Returns None

References

[R2], *[R3]*

broydens_second_method

odl.solvers.findroot.newton.**broydens_second_method**(*grad*, *x*, *line_search*, *niter=1*, *partial=None*)

Broyden's first method, a quasi-Newton scheme.

This is a general and optimized implementation of Broyden's second (or 'bad') method, a quasi-Newton method for solving a general unconstrained optimization problem

$$\min f(x)$$

for a differentiable function $f : \mathcal{X} \to \mathbb{R}$ on a Hilbert space $\mathcal{X}$. It does so by finding a zero of the gradient

$$\nabla f : \mathcal{X} \to \mathcal{X}$$

using a Newton-type update scheme with approximate Hessian.

The algorithm is described in *[R4]* and *[R5]*, and in a Wikipedia article

Parametersgrad : *Operator*

Gradient mapping of the objective function, i.e. the mapping $x \mapsto \nabla f(x) \in \mathcal{X}$

x : element in the domain of grad

Starting point of the iteration

line_search : *LineSearch*

Strategy to choose the step length

niter : int, optional

Number of iterations

partial : *Partial*, optional

Object executing code per iteration, e.g. plotting each iterate

References

[R4], *[R5]*

iterative

Modules

iterative

Simple iterative type optimization schemes.

Functions

conjugate_gradient(op, x, rhs[, niter, partial])	Optimized implementation of CG for self-adjoint operators.
conjugate_gradient_normal(op, x, rhs[, ...])	Optimized implementation of CG for the normal equation.
exp_zero_seq(base)	The default exponential zero sequence.
gauss_newton(op, x, rhs[, niter, zero_seq, ...])	Optimized implementation of a Gauss-Newton method.
landweber(op, x, rhs[, niter, omega, partial])	Optimized implementation of Landweber's method.

conjugate_gradient

odl.solvers.iterative.iterative.**conjugate_gradient**(*op*, *x*, *rhs*, *niter=1*, *partial=None*)

Optimized implementation of CG for self-adjoint operators.

This method solves the inverse problem (of the first kind)

$Ax = y$

for a linear and self-adjoint *Operator* A.

It uses a minimum amount of memory copies by applying re-usable temporaries and in-place evaluation.

The method is described (for linear systems) in a Wikipedia article.

Parametersop : linear *Operator*

Operator in the inverse problem. It must be linear and self-adjoint. This implies in particular that its domain and range are equal.

x : element of the domain of op

Vector to which the result is written. Its initial value is used as starting point of the iteration, and its values are updated in each iteration step.

rhs : element of the range of op

Right-hand side of the equation defining the inverse problem

niter : int, optional

Maximum number of iterations

partial : *Partial*, optional

Object executing code per iteration, e.g. plotting each iterate

ReturnsNone

conjugate_gradient_normal

odl.solvers.iterative.iterative.**conjugate_gradient_normal**(*op*, *x*, *rhs*, *niter=1*, *partial=None*)

Optimized implementation of CG for the normal equation.

This method solves the normal equation

$A^*Ax = A^*y$

to the inverse problem (of the first kind)

$Ax = y$

with a linear *Operator* A.

It uses a minimum amount of memory copies by applying re-usable temporaries and in-place evaluation.

The method is described (for linear systems) in a Wikipedia article.

Parametersop : *Operator*

Operator in the inverse problem. If not linear, it must have an implementation of *Operator.derivative*, which in turn must implement *Operator.adjoint*, i.e. the call `op.derivative(x).adjoint` must be valid.

x : element of the domain of `op`

Vector to which the result is written. Its initial value is used as starting point of the iteration, and its values are updated in each iteration step.

rhs : element of the range of `op`

Right-hand side of the equation defining the inverse problem

niter : int, optional

Maximum number of iterations

partial : *Partial*, optional

Object executing code per iteration, e.g. plotting each iterate

Returns`None`

exp_zero_seq

`odl.solvers.iterative.iterative.`**`exp_zero_seq`**(*base*)

The default exponential zero sequence.

It is defined by

t_0 = 1.0 t_m = t_(m-1) / base

or, in closed form

t_m = base^(-m-1)

Parametersbase : `float`

Base of the sequence. Its absolute value must be larger than 1.

Yieldsval : `float`

The next value in the exponential sequence

gauss_newton

`odl.solvers.iterative.iterative.`**`gauss_newton`**(*op*, *x*, *rhs*, *niter=1*, *zero_seq=<generator object exp_zero_seq>*, *partial=None*)

Optimized implementation of a Gauss-Newton method.

This method solves the inverse problem (of the first kind)

$A(x) = y$

for a (Frechet-) differentiable *Operator* `A` using a Gauss-Newton iteration.

It uses a minimum amount of memory copies by applying re-usable temporaries and in-place evaluation.

A variant of the method applied to a specific problem is described in a Wikipedia article.

Parametersop : *Operator*

Operator in the inverse problem. If not linear, it must have an implementation of *Operator.derivative*, which in turn must implement *Operator.adjoint*, i.e. the call `op.derivative(x).adjoint` must be valid.

x : element of the domain of `op`

Vector to which the result is written. Its initial value is used as starting point of the iteration, and its values are updated in each iteration step.

rhs : element of the range of `op`

Right-hand side of the equation defining the inverse problem

niter : `int`, optional

Maximum number of iterations

zero_seq : `iterable`, optional

Zero sequence whose values are used for the regularization of the linearized problem in each Newton step

partial : *Partial*, optional

Object executing code per iteration, e.g. plotting each iterate

ReturnsNone

landweber

odl.solvers.iterative.iterative.**landweber**(*op*, *x*, *rhs*, *niter=1*, *omega=1*, *partial=None*)

Optimized implementation of Landweber's method.

This method calculates an approximate least-squares solution of the inverse problem of the first kind

$$\mathcal{A}(x) = y,$$

for a given $y \in \mathcal{Y}$, i.e. an approximate solution x^* to

$$\min_{x \in \mathcal{X}} \|\mathcal{A}(x) - y\|_{\mathcal{Y}}^2$$

for a (Frechet-) differentiable operator $\mathcal{A} : \mathcal{X} \to \mathcal{Y}$ between Hilbert spaces $\mathcal{X}$ and $\mathcal{Y}$. The method starts from an initial guess x_0 and uses the iteration

$$x_{k+1} = x_k - \omega \, \partial\mathcal{A}(x)^*(\mathcal{A}(x_k) - y),$$

where $\partial\mathcal{A}(x)$ is the Frechet derivativ of $\mathcal{A}$ at x and ω is a relaxation parameter. For linear problems, a choice $0 < \omega < 2/\|\mathcal{A}\|$ guarantees convergence, where $\|\mathcal{A}\|$ stands for the operator norm of $\mathcal{A}$.

This implementation uses a minimum amount of memory copies by applying re-usable temporaries and in-place evaluation.

The method is also described in a Wikipedia article.

Parametersop : *Operator*

Operator in the inverse problem. It must have a *Operator.derivative* property, which returns a new operator which in turn has an *Operator.adjoint* property, i.e. `op.derivative(x).adjoint` must be well-defined for `x` in the operator domain.

x : element of the domain of `op`

Vector to which the result is written. Its initial value is used as starting point of the iteration, and its values are updated in each iteration step.

rhs : element of the range of `op`

Right-hand side of the equation defining the inverse problem

niter : `int`, optional

Maximum number of iterations

omega : positive `float`

Relaxation parameter in the iteration

partial : *Partial*, optional

Object executing code per iteration, e.g. plotting each iterate

Returns `None`

linear

scalar

Gradient-based optimization schemes.

Modules

gradient

Gradient-based optimization schemes.

Functions

steepest_descent(grad, x, line_search[, ...])	Steepest descent method to minimize an objective function.

steepest_descent

`odl.solvers.scalar.gradient.`**`steepest_descent`**(*grad*, *x*, *line_search*, *niter=1*, *partial=None*)

Steepest descent method to minimize an objective function.

General implementation of steepest decent (also known as gradient decent) for solving

$$\min f(x)$$

The algorithm is intended for unconstrained problems. It needs line search in order guarantee convergence. With appropriate line search, it can also be used for constrained problems where one wants to minimize over some given set C. This is done by defining $f(x) = \infty$ for $x \notin C$.

The algorithm is described in *[R6]*, section 9.3–9.4 (book available online), *[R7]*, Section 12.2, and wikipedia Gradient_descent.

Parametersgrad : *Operator*

Gradient of the objective function, $x \mapsto \nabla f(x)$

x : element in the domain of `deriv`

Starting point of the iteration

line_search : *LineSearch*

Strategy to choose the step length

niter : `int`, optional

Number of iterations

partial : *Partial*, optional

Object executing code per iteration, e.g. plotting each iterate

References

[R6], *[R7]*

steplen

Step length computation for optimization schemes.

Classes

BacktrackingLineSearch(function[, tau, c, ...])	Backtracking line search for step length calculation.
BarzilaiBorweinStep(gradf[, step0])	Barzilai-Borwein method to compute a step length.
ConstantLineSearch(constant)	Line search object that returns a constant step length.
LineSearch	Abstract base class for line search step length methods.
StepLength	Abstract base class for step length methods.

BacktrackingLineSearch

class `odl.solvers.scalar.steplen.`**`BacktrackingLineSearch`**(*function*, *tau=0.5*, *c=0.01*, *max_num_iter=None*)

Bases: *odl.solvers.scalar.steplen.LineSearch*

Backtracking line search for step length calculation.

This methods approximately finds the longest step length fulfilling the Armijo-Goldstein condition.

The line search algorithm is described in *[R8]*, page 464 (book available online) and *[R9]*, pages 378–379. See also Backtracking_line_search.

References

[R8], *[R9]*

Methods

__call__(x, direction, dir_derivative)	Calculate the optimal step length along a line.
__eq__	Return self==value.

BacktrackingLineSearch.__call__

`BacktrackingLineSearch.`**`__call__`**(*x*, *direction*, *dir_derivative*)

Calculate the optimal step length along a line.

Parametersx : *Operator.domain* element

The current point

direction : *Operator.domain* element

Search direction in which the line search should be computed

dir_derivative : float

Directional derivative along the direction

Returnsstep : float

The computed step length

__init__(*function*, *tau=0.5*, *c=0.01*, *max_num_iter=None*)

Initialize a new instance.

Parametersfunction : callable

The cost function of the optimization problem to be solved.

tau : float, optional

The amount the step length is decreased in each iteration, as long as it does not fulfill the decrease condition. The step length is updated as `step_length *= tau`

c : float, optional

The 'discount factor' on the `step length * direction derivative`, which the new point needs to be smaller than in order to fulfill the condition and be accepted (see the references).

max_num_iter : int, optional

Maximum number of iterations allowed each time the line search method is called. If not set, this number is calculated to allow a shortest step length of 0.0001.

BarzilaiBorweinStep

class odl.solvers.scalar.steplen.**BarzilaiBorweinStep**(*gradf*, *step0=0.0005*)

Bases: object

Barzilai-Borwein method to compute a step length.

Barzilai-Borwein method to compute a step length for gradient descent methods.

TODO

References

[R10], *[R11]*

Methods

__call__(x, x0)	Calculate the step length at a point.
__eq__	Return self==value.

BarzilaiBorweinStep.__call__

BarzilaiBorweinStep.**__call__**(*x*, *x0*)

Calculate the step length at a point.

Parametersx : *Operator.domain* element

The current point

x0 : *Operator.domain* element

The previous point

Returnsstep : float

The step length

__init__(*gradf*, *step0=0.0005*)

Initialize a new instance.

Parametersgradf : *Operator*

The gradient of the objective function at a point

step0 : float, optional

Initial steplength parameter

ConstantLineSearch

class odl.solvers.scalar.steplen.**ConstantLineSearch**(*constant*)

Bases: *odl.solvers.scalar.steplen.LineSearch*

Line search object that returns a constant step length.

Methods

__call__(x, direction, dir_derivative)	Calculate the step length at a point.
__eq__	Return self==value.

ConstantLineSearch.__call__

ConstantLineSearch.**__call__**(*x*, *direction*, *dir_derivative*)

Calculate the step length at a point.

Parametersx : *Operator.domain* element

The current point

direction : *Operator.domain* element

Search direction in which the line search should be computed

dir_derivative : float

Directional derivative along the direction

Returnsstep : float

The constant step length

__init__(*constant*)

Initialize a new instance.

Parametersconstant : float

The constant step length

LineSearch

class odl.solvers.scalar.steplen.**LineSearch**
Bases: object

Abstract base class for line search step length methods.

Methods

__call__(x, direction, dir_derivative)	Calculate step length in direction.
__eq__	Return self==value.

LineSearch.__call__
LineSearch.**__call__**(*x*, *direction*, *dir_derivative*)
Calculate step length in direction.

Parametersx : *Operator.domain* element

The current point

direction : *Operator.domain* element

Search direction in which the line search should be computed

dir_derivative : float

Directional derivative along the direction

Returnsstep : float

The step length

StepLength
class odl.solvers.scalar.steplen.**StepLength**
Bases: object

Abstract base class for step length methods.

Methods

__call__(x, direction, dir_derivative)	Calculate the step length at a point.
__eq__	Return self==value.

StepLength.__call__
StepLength.**__call__**(*x*, *direction*, *dir_derivative*)
Calculate the step length at a point.

Parametersx : *Operator.domain* element

The current point

direction : *Operator.domain* element

Search direction in which the line search should be computed

dir_derivative : float

Directional derivative along the direction

Returnsstep : `float`

The step length

util

Modules

partial

Classes

ForEachPartial(function)	Simple object for applying a function to each iterate.
Partial	Abstract base class for sending partial results of iterations.
PrintIterationPartial()	Print the interation count.
PrintStatusPartial()	Print the interation count and current norm of each iterate.
StorePartial()	Simple object for storing all partial results of the solvers.

ForEachPartial

class `odl.solvers.util.partial.`**`ForEachPartial`**(*function*)

Bases: *odl.solvers.util.partial.Partial*

Simple object for applying a function to each iterate.

Methods

`__eq__`	Return self==value.
send(result)	Applies function to result.

ForEachPartial.send

`ForEachPartial.`**`send`**(*result*)

Applies function to result.

`__init__`(*function*)

Partial

class `odl.solvers.util.partial.`**`Partial`**

Bases: `object`

Abstract base class for sending partial results of iterations.

Methods

`__eq__`	Return self==value.
send(result)	Send the result to the partial object.

Partial.send

`Partial.`**`send`**(*result*)

Send the result to the partial object.

PrintIterationPartial

class odl.solvers.util.partial.**PrintIterationPartial**

Bases: *odl.solvers.util.partial.Partial*

Print the interation count.

Methods

__eq__	Return self==value.
send(_)	Print the current iteration.

PrintIterationPartial.send

PrintIterationPartial.**send**(_)

Print the current iteration.

__init__()

PrintStatusPartial

class odl.solvers.util.partial.**PrintStatusPartial**

Bases: *odl.solvers.util.partial.Partial*

Print the interation count and current norm of each iterate.

Methods

__eq__	Return self==value.
send(result)	Print the current iteration and norm.

PrintStatusPartial.send

PrintStatusPartial.**send**(*result*)

Print the current iteration and norm.

__init__()

StorePartial

class odl.solvers.util.partial.**StorePartial**

Bases: *odl.solvers.util.partial.Partial*

Simple object for storing all partial results of the solvers.

Methods

__eq__	Return self==value.
send(result)	Append result to results list.

StorePartial.send

StorePartial.**send**(*result*)

Append result to results list.

__init__()

vector

Modules

newton

Newton type optimization schemes.

Functions

newtons_method(op, x, line_search[, ...])	Newton's method for solving a system of equations.

newtons_method

odl.solvers.vector.newton.**newtons_method**(*op*, *x*, *line_search*, *num_iter=10*, *cg_iter=None*, *partial=None*)

Newton's method for solving a system of equations.

This is a general and optimized implementation of Newton's method for solving the problem

$$f(x) = 0.$$

of finding a root of a function f.

The algorithm is well-known and there is a vast literature about it. Among others, the method is described in *[R12]*, Sections 9.5 and 10.2 (book available online), *[R13]*, Section 2.7 for solving nonlinear equations and Section 11.3 for it's use in minimization, and wikipedia on Newton's_method.

Parametersderiv : *Operator*

Gradient of the objective function, $x \mapsto gradf(x)$

x : element in the domain of deriv

Starting point of the iteration

line_search : *LineSearch*

Strategy to choose the step length

num_iter : int, optional

Number of iterations

cg_iter : int, optional

Number of iterations in the the conjugate gradient solver, for comuting the search direction.

partial : *Partial*, optional

Object executing code per iteration, e.g. plotting each iterate

Notes

The algorithm works by iteratively solving

$$\partial f(x_k) p_k = -f(x_k)$$

and then updating as

$$x_{k+1} = x_k + \alpha x_k,$$

where α is a suitable step length (see the references). In this implementation the system of equations are solved using the conjugate gradient method.

References

[R12], *[R13]*

2.4.6 space

Concrete vector spaces.

Modules

base_ntuples

Base classes for implementation of n-tuples.

Classes

FnBase(size, dtype)	Base class for F^n independent of implementation.
FnBaseVector(space, *args, **kwargs)	Abstract class for representation of *FnBase* vectors.
FnWeightingBase(impl[, exponent, ...])	Abstract base class for weighting of *FnBase* spaces.
NtuplesBase(size, dtype)	Base class for sets of n-tuples independent of implementation.
NtuplesBaseVector(space, *args, **kwargs)	Abstract class for representation of *NtuplesBase* elements.

FnBase

class odl.space.base_ntuples.**FnBase**(*size*, *dtype*)

Bases: *odl.space.base_ntuples.NtuplesBase*, *odl.set.space.LinearSpace*

Base class for F^n independent of implementation.

Attributes

dtype	The data type of each entry.
element_type	*FnBaseVector*
field	The field of this vector space.
is_cn	If the space represents the set C^n.
is_rn	If the space represents the set R^n.
real_dtype	The corresponding real data type of this space.
size	The number of entries per tuple.

FnBase.dtype
`FnBase.dtype`
The data type of each entry.

FnBase.element_type
`FnBase.element_type`
FnBaseVector

FnBase.field
`FnBase.field`
The field of this vector space.

The field is the set of scalars of the space, that is numbers that the vectors in the space can be multiplied with.

Returnsfield : *Field*

The underlying field.

FnBase.is_cn
`FnBase.is_cn`
If the space represents the set C^n.

Tuples of complex numbers.

FnBase.is_rn
`FnBase.is_rn`
If the space represents the set R^n.

Tuples of real numbers.

FnBase.real_dtype
`FnBase.real_dtype`
The corresponding real data type of this space.

FnBase.size
`FnBase.size`
The number of entries per tuple.

Methods

__contains__(other)	Return `other in self`.
__eq__(other)	Return `self == other`.
_divide(x1, x2, out)	The entry-wise division of two vectors, assigned to `out`.
_multiply(x1, x2, out)	The entry-wise product of two vectors, assigned to `out`.
one()	Create a vector of ones.
zero()	Create a vector of zeros.

FnBase.__contains__
`FnBase.__contains__`(*other*)

Return `other in self`.

Returnscontains : `bool`

`True` if `other` is an *`NtuplesBaseVector`* instance and `other.space` is equal to this space, `False` otherwise.

Examples

```
>>> from odl import Ntuples
>>> long_3 = Ntuples(3, dtype='int64')
>>> long_3.element() in long_3
True
>>> long_3.element() in Ntuples(3, dtype='int32')
False
>>> long_3.element() in Ntuples(3, dtype='float64')
False
```

FnBase.__eq__

`FnBase.__eq__`(*other*)

Return `self == other`.

Returnsequals : `bool`

`True` if `other` is an instance of this space's type with the same *`size`* and *`dtype`*, otherwise `False`.

Examples

```
>>> from odl import Ntuples
>>> int_3 = Ntuples(3, dtype=int)
>>> int_3 == int_3
True
```

Equality is not identity:

```
>>> int_3a, int_3b = Ntuples(3, int), Ntuples(3, int)
>>> int_3a == int_3b
True
>>> int_3a is int_3b
False
```

```
>>> int_3, int_4 = Ntuples(3, int), Ntuples(4, int)
>>> int_3 == int_4
False
>>> int_3, str_3 = Ntuples(3, 'int'), Ntuples(3, 'S2')
>>> int_3 == str_3
False
```

FnBase._divide

`FnBase._divide`(*x1*, *x2*, *out*)

The entry-wise division of two vectors, assigned to `out`.

FnBase._multiply
`FnBase.`**`_multiply`**(*x1*, *x2*, *out*)
The entry-wise product of two vectors, assigned to `out`.

FnBase.one
`FnBase.`**`one`**`()`
Create a vector of ones.

FnBase.zero
`FnBase.`**`zero`**`()`
Create a vector of zeros.
`__init__`(*size*, *dtype*)
Initialize a new instance.

Parameterssize : `int`

The number of dimensions of the space

dtype : `object`

The data type of the storage array. Can be provided in any way the `numpy.dtype` function understands, most notably as built-in type, as one of NumPy's internal datatype objects or as string. Only scalar data types (numbers) are allowed.

FnBaseVector

class `odl.space.base_ntuples.`**`FnBaseVector`**(*space*, **args*, ***kwargs*)
Bases: *odl.space.base_ntuples.NtuplesBaseVector*, *odl.set.space.LinearSpaceVector*

Abstract class for representation of *FnBase* vectors.

Defines abstract attributes and concrete ones which are independent of data representation.

Attributes

T	The transpose of a vector, the functional given by (.
dtype	Length of this vector, equal to space size.
itemsize	The size in bytes on one element of this type.
nbytes	The number of bytes this vector uses in memory.
ndim	Number of dimensions, always 1.
shape	Shape of this vector, equals `(size,)`.
size	Length of this vector, equal to space size.
space	Space to which this vector.
ufunc	*NtuplesBaseUFuncs*, access to numpy style ufuncs.

FnBaseVector.T
`FnBaseVector.`**`T`**
The transpose of a vector, the functional given by (. , self)

Returnstranspose : *InnerProductOperator*

Notes

This function is only defined in inner product spaces.

In a complex space, this takes the conjugate transpose of the vector.

Examples

```
>>> from odl import Rn
>>> import numpy as np
>>> rn = Rn(3)
>>> x = rn.element([1, 2, 3])
>>> y = rn.element([2, 1, 3])
>>> x.T(y)
13.0
```

FnBaseVector.dtype

`FnBaseVector.`**`dtype`**

Length of this vector, equal to space size.

FnBaseVector.itemsize

`FnBaseVector.`**`itemsize`**

The size in bytes on one element of this type.

FnBaseVector.nbytes

`FnBaseVector.`**`nbytes`**

The number of bytes this vector uses in memory.

FnBaseVector.ndim

`FnBaseVector.`**`ndim`**

Number of dimensions, always 1.

FnBaseVector.shape

`FnBaseVector.`**`shape`**

Shape of this vector, equals `(size,)`.

FnBaseVector.size

`FnBaseVector.`**`size`**

Length of this vector, equal to space size.

FnBaseVector.space

`FnBaseVector.`**`space`**

Space to which this vector.

FnBaseVector.ufunc

`FnBaseVector.`**`ufunc`**

`NtuplesBaseUFuncs`, access to numpy style ufuncs.

These are always available, but may or may not be optimized for the specific space in use.

Methods

__eq__(other)	
__getitem__(indices)	Access values of this vector.
__setitem__(indices, values)	Set values of this vector.
asarray([start, stop, step, out])	Extract the data of this array as a numpy array.
copy()	

FnBaseVector.__eq__
`FnBaseVector.__eq__`(*other*)

FnBaseVector.__getitem__
`FnBaseVector.__getitem__`(*indices*)

Access values of this vector.

Parametersindices : `int` or `slice`

The position(s) that should be accessed

Returnsvalues : *NtuplesBase.dtype* or *NtuplesBaseVector*

The value(s) at the index (indices)

FnBaseVector.__setitem__
`FnBaseVector.__setitem__`(*indices*, *values*)

Set values of this vector.

Parametersindices : `int` or `slice`

The position(s) that should be set

values : {scalar, array-like, *NtuplesBaseVector*}

The value(s) that are to be assigned.

If `index` is an integer, `value` must be single value.

If `index` is a slice, `value` must be broadcastable to the size of the slice (same size, shape (1,) or single value).

FnBaseVector.asarray
`FnBaseVector.asarray`(*start=None*, *stop=None*, *step=None*, *out=None*)

Extract the data of this array as a numpy array.

Parametersstart : `int`, optional

Start position. `None` means the first element.

start : `int`, optional

One element past the last element to be extracted. `None` means the last element.

start : `int`, optional

Step length. `None` means 1.

out : `numpy.ndarray`

Array to write result to.

Returnsasarray : `numpy.ndarray`

Numpy array of the same type as the space.

FnBaseVector.copy
`FnBaseVector.`**`copy`**`()`
`__init__`(*space*, **args*, ***kwargs*)
Initialize a new instance.

FnWeightingBase

class `odl.space.base_ntuples.`**`FnWeightingBase`**(*impl*, *exponent=2.0*, *dist_using_inner=False*)
Bases: `object`

Abstract base class for weighting of *`FnBase`* spaces.

This class and its subclasses serve as a simple means to evaluate and compare weighted inner products, norms and metrics semantically rather than by identity on a pure function level.

The functions are implemented similarly to *`Operator`*, but without extra type checks of input parameters - this is done in the callers of the *`LinearSpace`* instance where these functions used.

Attributes

`exponent`	Exponent of this weighting.
`impl`	Implementation backend of this weighting.

FnWeightingBase.exponent
`FnWeightingBase.`**`exponent`**
Exponent of this weighting.

FnWeightingBase.impl
`FnWeightingBase.`**`impl`**
Implementation backend of this weighting.

Methods

`__eq__`(other)	Return `self == other`.
`dist`(x1, x2)	Calculate the distance between two vectors.
`equiv`(other)	Test if `other` is an equivalent inner product.
`inner`(x1, x2)	Calculate the inner product of two vectors.
`norm`(x)	Calculate the norm of a vector.

FnWeightingBase.__eq__
`FnWeightingBase.`**`__eq__`**(*other*)
Return `self == other`.

Returnsequal : `bool`

`True` if `other` is a the same weighting, `False` otherwise.

Notes

This operation must be computationally cheap, i.e. no large arrays may be compared element-wise. That is the task of the *equiv* method.

FnWeightingBase.dist
FnWeightingBase.**dist**(*x1*, *x2*)
Calculate the distance between two vectors.

This is the standard implementation using *norm*. Subclasses should override it for optimization purposes.

Parametersx1, x2 : *FnBaseVector*

Vectors whose mutual distance is calculated

Returnsdist : float

The distance between the vectors

FnWeightingBase.equiv
FnWeightingBase.**equiv**(*other*)
Test if other is an equivalent inner product.

Should be overwritten, default tests for equality.

Returnsequivalent : bool

True if other is a *FnWeightingBase* instance which yields the same result as this inner product for any input, False otherwise.

FnWeightingBase.inner
FnWeightingBase.**inner**(*x1*, *x2*)
Calculate the inner product of two vectors.

Parametersx1, x2 : *FnBaseVector*

Vectors whose inner product is calculated

Returnsinner : float or complex

The inner product of the two provided vectors

FnWeightingBase.norm
FnWeightingBase.**norm**(*x*)
Calculate the norm of a vector.

This is the standard implementation using *inner*. Subclasses should override it for optimization purposes.

Parametersx1 : *FnBaseVector*

Vector whose norm is calculated

Returnsnorm : float

The norm of the vector
__init__(*impl*, *exponent=2.0*, *dist_using_inner=False*)
Initialize a new instance.

Parametersimpl : str

Specifier for the implementation backend

exponent : positive `float`

Exponent of the norm. For values other than 2.0, the inner product is not defined.

dist_using_inner : `bool`, optional

Calculate *dist* using the formula

$\|x - y\|^2 = \|x\|^2 + \|y\|^2 - 2\Re\langle x, y\rangle$.

This avoids the creation of new arrays and is thus faster for large arrays. On the downside, it will not evaluate to exactly zero for equal (but not identical) x and y.

This option can only be used if `exponent` is 2.0.

Default: `False`.

NtuplesBase

class `odl.space.base_ntuples.`**`NtuplesBase`**(*size*, *dtype*)

Bases: *odl.set.sets.Set*

Base class for sets of n-tuples independent of implementation.

Attributes

dtype	The data type of each entry.
element_type	*NtuplesBaseVector*
size	The number of entries per tuple.

NtuplesBase.dtype

`NtuplesBase.`**`dtype`**

The data type of each entry.

NtuplesBase.element_type

`NtuplesBase.`**`element_type`**

NtuplesBaseVector

NtuplesBase.size

`NtuplesBase.`**`size`**

The number of entries per tuple.

Methods

__contains__(other)	Return `other in self`.
__eq__(other)	Return `self == other`.

NtuplesBase.__contains__

`NtuplesBase.`**`__contains__`**(*other*)

Return `other in self`.

Returnscontains : `bool`

`True` if `other` is an *`NtuplesBaseVector`* instance and `other.space` is equal to this space, `False` otherwise.

Examples

```
>>> from odl import Ntuples
>>> long_3 = Ntuples(3, dtype='int64')
>>> long_3.element() in long_3
True
>>> long_3.element() in Ntuples(3, dtype='int32')
False
>>> long_3.element() in Ntuples(3, dtype='float64')
False
```

NtuplesBase.__eq__

`NtuplesBase.`**`__eq__`**`(`*other*`)`

Return `self == other`.

Returnsequals : `bool`

`True` if `other` is an instance of this space's type with the same *`size`* and *`dtype`*, otherwise `False`.

Examples

```
>>> from odl import Ntuples
>>> int_3 = Ntuples(3, dtype=int)
>>> int_3 == int_3
True
```

Equality is not identity:

```
>>> int_3a, int_3b = Ntuples(3, int), Ntuples(3, int)
>>> int_3a == int_3b
True
>>> int_3a is int_3b
False
```

```
>>> int_3, int_4 = Ntuples(3, int), Ntuples(4, int)
>>> int_3 == int_4
False
>>> int_3, str_3 = Ntuples(3, 'int'), Ntuples(3, 'S2')
>>> int_3 == str_3
False
```

`__init__``(`*size*, *dtype*`)`

Initialize a new instance.

Parameterssize : non-negative int

The number of entries per tuple

dtype : `object`

The data type for each tuple entry. Can be provided in any way the `numpy.dtype` function understands, most notably as built-in type, as one of NumPy's internal datatype objects or as string.

NtuplesBaseVector

class `odl.space.base_ntuples.`**`NtuplesBaseVector`**(*space*, **args*, ***kwargs*)
Bases: `object`

Abstract class for representation of *`NtuplesBase`* elements.

Defines abstract attributes and concrete ones which are independent of data representation.

Attributes

`dtype`	Length of this vector, equal to space size.
`itemsize`	The size in bytes on one element of this type.
`nbytes`	The number of bytes this vector uses in memory.
`ndim`	Number of dimensions, always 1.
`shape`	Shape of this vector, equals `(size,)`.
`size`	Length of this vector, equal to space size.
`space`	Space to which this vector.
`ufunc`	*`NtuplesBaseUFuncs`*, access to numpy style ufuncs.

NtuplesBaseVector.dtype
`NtuplesBaseVector.`**`dtype`**
Length of this vector, equal to space size.

NtuplesBaseVector.itemsize
`NtuplesBaseVector.`**`itemsize`**
The size in bytes on one element of this type.

NtuplesBaseVector.nbytes
`NtuplesBaseVector.`**`nbytes`**
The number of bytes this vector uses in memory.

NtuplesBaseVector.ndim
`NtuplesBaseVector.`**`ndim`**
Number of dimensions, always 1.

NtuplesBaseVector.shape
`NtuplesBaseVector.`**`shape`**
Shape of this vector, equals `(size,)`.

NtuplesBaseVector.size
`NtuplesBaseVector.`**`size`**
Length of this vector, equal to space size.

NtuplesBaseVector.space
NtuplesBaseVector.**space**
Space to which this vector.

NtuplesBaseVector.ufunc
NtuplesBaseVector.**ufunc**
NtuplesBaseUFuncs, access to numpy style ufuncs.

These are always available, but may or may not be optimized for the specific space in use.

Methods

__eq__(other)	Return `self == other`.
__getitem__(indices)	Access values of this vector.
__setitem__(indices, values)	Set values of this vector.
asarray([start, stop, step, out])	Extract the data of this array as a numpy array.
copy()	Create an identical (deep) copy of this vector.

NtuplesBaseVector.__eq__
NtuplesBaseVector.**__eq__**(*other*)
Return `self == other`.

Returnsequals : `bool`

`True` if all entries of `other` are equal to this vector's entries, `False` otherwise.

NtuplesBaseVector.__getitem__
NtuplesBaseVector.**__getitem__**(*indices*)
Access values of this vector.

Parametersindices : `int` or `slice`

The position(s) that should be accessed

Returnsvalues : *NtuplesBase.dtype* or *NtuplesBaseVector*

The value(s) at the index (indices)

NtuplesBaseVector.__setitem__
NtuplesBaseVector.**__setitem__**(*indices*, *values*)
Set values of this vector.

Parametersindices : `int` or `slice`

The position(s) that should be set

values : {scalar, array-like, *NtuplesBaseVector*}

The value(s) that are to be assigned.

If `index` is an integer, `value` must be single value.

If `index` is a slice, `value` must be broadcastable to the size of the slice (same size, shape (1,) or single value).

NtuplesBaseVector.asarray
`NtuplesBaseVector.`**`asarray`**`(`*start=None*, *stop=None*, *step=None*, *out=None*`)`
Extract the data of this array as a numpy array.

Parametersstart : `int`, optional

Start position. `None` means the first element.

start : `int`, optional

One element past the last element to be extracted. `None` means the last element.

start : `int`, optional

Step length. `None` means 1.

out : `numpy.ndarray`

Array to write result to.

Returnsasarray : `numpy.ndarray`

Numpy array of the same type as the space.

NtuplesBaseVector.copy
`NtuplesBaseVector.`**`copy`**`()`
Create an identical (deep) copy of this vector.
`__init__``(`*space*, **args*, ***kwargs*`)`
Initialize a new instance.

cu_ntuples

CUDA implementation of n-dimensional Cartesian spaces.

Classes

CudaFn(size, dtype, **kwargs)	The space *FnBase*, implemented in CUDA.
CudaFnConstWeighting(constant[, exponent])	Weighting of *CudaFn* by a constant.
CudaFnCustomDist(dist)	Custom distance on *CudaFn*, removes `norm` and `inner`.
CudaFnCustomInnerProduct(inner[, ...])	Custom inner product on *CudaFn*.
CudaFnCustomNorm(norm)	Custom norm on *CudaFn*, removes `inner`.
CudaFnNoWeighting([exponent])	Weighting of *CudaFn* with constant 1.
CudaFnVector(space, data)	Representation of a *CudaFn* element.
CudaFnVectorWeighting(vector[, exponent])	Vector weighting for *CudaFn*.
CudaNtuples(size, dtype)	The space *NtuplesBase*, implemented in CUDA.
CudaNtuplesVector(space, data)	Representation of a *CudaNtuples* element.

CudaFn

class `odl.space.cu_ntuples.`**`CudaFn`**`(`*size*, *dtype*, ***kwargs*`)`
Bases: *odl.space.base_ntuples.FnBase*, *odl.space.cu_ntuples.CudaNtuples*

The space *FnBase*, implemented in CUDA.

Requires the compiled ODL extension odlpp.

Attributes

dtype	The data type of each entry.
element_type	*CudaFnVector*
exponent	Exponent of the norm and distance.
field	The field of this vector space.
is_cn	If the space represents the set C^n.
is_rn	If the space represents the set R^n.
real_dtype	The corresponding real data type of this space.
size	The number of entries per tuple.

CudaFn.dtype
CudaFn.**dtype**
The data type of each entry.

CudaFn.element_type
CudaFn.**element_type**
CudaFnVector

CudaFn.exponent
CudaFn.**exponent**
Exponent of the norm and distance.

CudaFn.field
CudaFn.**field**
The field of this vector space.

The field is the set of scalars of the space, that is numbers that the vectors in the space can be multiplied with.

Returnsfield : *Field*

The underlying field.

CudaFn.is_cn
CudaFn.**is_cn**
If the space represents the set C^n.

Tuples of complex numbers.

CudaFn.is_rn
CudaFn.**is_rn**
If the space represents the set R^n.

Tuples of real numbers.

CudaFn.real_dtype
CudaFn.**real_dtype**
The corresponding real data type of this space.

CudaFn.size

`CudaFn.size`

The number of entries per tuple.

Methods

__eq__(other)	s.__eq__(other) <==> s == other.
_dist(x1, x2)	Calculate the distance between two vectors.
_divide(x1, x2, out)	The pointwise division of two vectors, assigned to `out`.
_inner(x1, x2)	Calculate the inner product of x and y.
_lincomb(a, x1, b, x2, out)	Linear combination of `x1` and `x2`, assigned to `out`.
_multiply(x1, x2, out)	The pointwise product of two vectors, assigned to `out`.
_norm(x)	Calculate the norm of `x`.
element([inp, data_ptr])	Create a new element.
one()	Create a vector of ones.
zero()	Create a vector of zeros.

CudaFn.__eq__

`CudaFn.__eq__`(*other*)

s.__eq__(other) <==> s == other.

Returnsequals : `bool`

`True` if other is an instance of this space's type with the same `size`, `dtype` and space functions, otherwise `False`.

Examples

```
>>> from numpy.linalg import norm
>>> def dist(x, y, ord):
...     return norm(x - y, ord)
```

```
>>> from functools import partial
>>> dist2 = partial(dist, ord=2)
>>> r3 = CudaRn(3, dist=dist2)
>>> r3_same = CudaRn(3, dist=dist2)
>>> r3  == r3_same
True
```

Different `dist` functions result in different spaces - the same applies for `norm` and `inner`:

```
>>> dist1 = partial(dist, ord=1)
>>> r3_1 = CudaRn(3, dist=dist1)
>>> r3_2 = CudaRn(3, dist=dist2)
>>> r3_1 == r3_2
False
```

Be careful with Lambdas - they result in non-identical function objects:

```
>>> r3_lambda1 = CudaRn(3, dist=lambda x, y: norm(x-y, ord=1))
>>> r3_lambda2 = CudaRn(3, dist=lambda x, y: norm(x-y, ord=1))
>>> r3_lambda1 == r3_lambda2
False
```

CudaFn._dist

`CudaFn.`**`_dist`**(*x1*, *x2*)

Calculate the distance between two vectors.

Parametersx1, x2 : *CudaFnVector*

The vectors whose mutual distance is calculated

Returnsdist : `float`

Distance between the vectors

Examples

```
>>> r2 = CudaRn(2)
>>> x = r2.element([3, 8])
>>> y = r2.element([0, 4])
>>> r2.dist(x, y)
5.0
```

CudaFn._divide

`CudaFn.`**`_divide`**(*x1*, *x2*, *out*)

The pointwise division of two vectors, assigned to `out`.

This is defined as:

multiply(z, x, y) := [x[0]/y[0], x[1]/y[1], ..., x[n-1]/y[n-1]]

Parametersx1, x2 : *CudaFnVector*

Read from

out : *CudaFnVector*

Write to

ReturnsNone

Examples

```
>>> rn = CudaRn(3)
>>> x1 = rn.element([5, 3, 2])
>>> x2 = rn.element([1, 2, 2])
>>> out = rn.element()
>>> rn.divide(x1, x2, out)  # out is returned
CudaRn(3).element([5.0, 1.5, 1.0])
>>> out
CudaRn(3).element([5.0, 1.5, 1.0])
```

CudaFn._inner

`CudaFn.`**`_inner`**(*x1*, *x2*)

Calculate the inner product of x and y.

Parametersx1, x2 : *CudaFnVector*

Returnsinner: `float` or `complex`

The inner product of x and y

Examples

```
>>> uc3 = CudaFn(3, 'uint8')
>>> x = uc3.element([1, 2, 3])
>>> y = uc3.element([3, 1, 5])
>>> uc3.inner(x, y)
20.0
```

CudaFn._lincomb

`CudaFn.`**`_lincomb`**(*a*, *x1*, *b*, *x2*, *out*)

Linear combination of `x1` and `x2`, assigned to `out`.

Calculate `z = a * x + b * y` using optimized CUDA routines.

Parametersa, b : *LinearSpace.field* element

Scalar to multiply `x` and `y` with.

x, y : *CudaFnVector*

The summands

out : *CudaFnVector*

The Vector that the result is written to.

ReturnsNone

Examples

```
>>> r3 = CudaRn(3)
>>> x = r3.element([1, 2, 3])
>>> y = r3.element([4, 5, 6])
>>> out = r3.element()
>>> r3.lincomb(2, x, 3, y, out)  # out is returned
CudaRn(3).element([14.0, 19.0, 24.0])
>>> out
CudaRn(3).element([14.0, 19.0, 24.0])
```

CudaFn._multiply

`CudaFn.`**`_multiply`**(*x1*, *x2*, *out*)

The pointwise product of two vectors, assigned to `out`.

This is defined as:

multiply(x, y, out) := [x[0]*y[0], x[1]*y[1], ..., x[n-1]*y[n-1]]

Parametersx1, x2 : *CudaFnVector*

Factors in product

out : *CudaFnVector*

Result

ReturnsNone

Examples

```
>>> rn = CudaRn(3)
>>> x1 = rn.element([5, 3, 2])
>>> x2 = rn.element([1, 2, 3])
>>> out = rn.element()
>>> rn.multiply(x1, x2, out)  # out is returned
CudaRn(3).element([5.0, 6.0, 6.0])
>>> out
CudaRn(3).element([5.0, 6.0, 6.0])
```

CudaFn._norm

CudaFn.**_norm**(*x*)

Calculate the norm of `x`.

This method is implemented separately from `sqrt(inner(x,x))` for efficiency reasons.

Parametersx : *CudaFnVector*

Returnsnorm : `float`

The norm of x

Examples

```
>>> uc3 = CudaFn(3, 'uint8')
>>> x = uc3.element([2, 3, 6])
>>> uc3.norm(x)
7.0
```

CudaFn.element

CudaFn.**element**(*inp=None*, *data_ptr=None*)

Create a new element.

Parametersinp : array-like or scalar, optional

Input to initialize the new element.

If `inp` is a `numpy.ndarray` of shape `(size,)` and the same data type as this space, the array is wrapped, not copied. Other array-like objects are copied (with broadcasting if necessary).

If a single value is given, it is copied to all entries.

If both `inp` and `data_ptr` are `None`, an empty element is created with no guarantee of its state (memory allocation only).

data_ptr : `int`, optional

Memory address of a CUDA array container

Cannot be combined with `inp`.

Returnselement : *CudaNtuplesVector*

The new element

Notes

This method preserves "array views" of correct size and type, see the examples below.

TODO: No, it does not yet!

Examples

```
>>> uc3 = CudaNtuples(3, 'uint8')
>>> x = uc3.element(np.array([1, 2, 3], dtype='uint8'))
>>> x
CudaNtuples(3, 'uint8').element([1, 2, 3])
>>> y = uc3.element([1, 2, 3])
>>> y
CudaNtuples(3, 'uint8').element([1, 2, 3])
```

CudaFn.one

`CudaFn.`**`one`**`()`

Create a vector of ones.

CudaFn.zero

`CudaFn.`**`zero`**`()`

Create a vector of zeros.

`__init__`(*size*, *dtype*, ***kwargs*)

Initialize a new instance.

Parameters **size** : positive `int`

The number of dimensions of the space

dtype : `object`

The data type of the storage array. Can be provided in any way the `numpy.dtype` function understands, most notably as built-in type, as one of NumPy's internal datatype objects or as string.

Only scalar data types are allowed.

kwargs : {'weight', 'exponent', 'dist', 'norm', 'inner'}

'weight'[{array-like, *`CudaFnVector`*, `float`, `None`}] Use weighted inner product, norm, and dist.

`None`: No weighting, use standard functions (default)

`float`: Weighting by a constant

array-like: Weighting by a vector (1-dim. array, corresponds to a diagonal matrix). Note that the array is stored in main memory, which results in slower space functions due to a copy during evaluation.

***`CudaFnVector`*:**same as 1-dim. array-like, except that copying is avoided if the `dtype` of the vector is the same as this space's `dtype`.

This option cannot be combined with `dist`, `norm` or `inner`.

'exponent'[positive `float`] Exponent of the norm. For values other than 2.0, no inner product is defined.

This option is ignored if `dist`, `norm` or `inner` is given.

Default: 2.0

'dist'[`callable`, optional] The distance function defining a metric on $\mathbb{F}^n$. It must accept two *CudaFnVector* arguments, return a `float` and fulfill the following mathematical conditions for any three vectors x, y, z:

- $d(x, y) = d(y, x)$
- $d(x, y) \geq 0$
- $d(x, y) = 0 \Leftrightarrow x = y$
- $d(x, y) \geq d(x, z) + d(z, y)$

By default, `dist(x, y)` is calculated as `norm(x - y)`. This creates an intermediate array `x-y`, which can be avoided by choosing `dist_using_inner=True`.

This option cannot be combined with `weight`, `norm` or `inner`.

'norm'[`callable`, optional] The norm implementation. It must accept an *CudaFnVector* argument, return a `float` and satisfy the following conditions for all vectors x, y and scalars s:

- $\|x\| \geq 0$
- $\|x\| = 0 \Leftrightarrow x = 0$
- $\|sx\| = |s|\|x\|$
- $\|x + y\| \leq \|x\| + \|y\|$.

By default, `norm(x)` is calculated as `inner(x, x)`.

This option cannot be combined with `weight`, `dist` or `inner`.

'inner'[`callable`, optional] The inner product implementation. It must accept two *CudaFnVector* arguments, return a element from the field of the space (real or complex number) and satisfy the following conditions for all vectors x, y, z and scalars s:

- $\langle x, y\rangle = \overline{\langle y, x\rangle}$
- $\langle sx, y\rangle = s\langle x, y\rangle$
- $\langle x + z, y\rangle = \langle x, y\rangle + \langle z, y\rangle$
- $\langle x, x\rangle = 0 \Leftrightarrow x = 0$

This option cannot be combined with `weight`, `dist` or `norm`.

CudaFnConstWeighting

class `odl.space.cu_ntuples.`**`CudaFnConstWeighting`**(*constant*, *exponent=2.0*)

Bases: *odl.space.base_ntuples.FnWeightingBase*

Weighting of *CudaFn* by a constant.

For exponent 2.0, a new weighted inner product with constant c is defined as

$\langle a, b\rangle_c := c\, b^H a$

with b^H standing for transposed complex conjugate.

For other exponents, only norm and dist are defined. In the case of exponent `inf`, the weighted norm is defined as

$\|a\|_{c,\infty} := c\|a\|_\infty$,

otherwise it is

$\|a\|_{c,p} := c^{1/p}\|a\|_p$.

Not that this definition does **not** fulfill the limit property in p, i.e.

$\lim_{p\to\infty}\|a\|_{c,p} = \|a\|_\infty \neq \|a\|_{c,\infty}$

unless $c = 1$.

The constant c must be positive.

Attributes

const	Weighting constant of this inner product.
exponent	Exponent of this weighting.
impl	Implementation backend of this weighting.

CudaFnConstWeighting.const
`CudaFnConstWeighting.`**`const`**
Weighting constant of this inner product.

CudaFnConstWeighting.exponent
`CudaFnConstWeighting.`**`exponent`**
Exponent of this weighting.

CudaFnConstWeighting.impl
`CudaFnConstWeighting.`**`impl`**
Implementation backend of this weighting.

Methods

__eq__(other)	Return `self == other`.
dist(x1, x2)	Calculate the constant-weighted distance between two vectors.
equiv(other)	Test if `other` is an equivalent weighting.
inner(x1, x2)	Calculate the constant-weighted inner product of two vectors.
norm(x)	Calculate the constant-weighted norm of a vector.

CudaFnConstWeighting.__eq__
`CudaFnConstWeighting.`**`__eq__`**(*other*)
Return `self == other`.

Returnsequal : `bool`

`True` if `other` is a *CudaFnConstWeighting* instance with the same constant, `False` otherwise.

CudaFnConstWeighting.dist
`CudaFnConstWeighting.dist(x1, x2)`
Calculate the constant-weighted distance between two vectors.

Parametersx1, x2 : *CudaFnVector*

Vectors whose mutual distance is calculated

Returnsdist : `float`

The distance between the vectors

CudaFnConstWeighting.equiv
`CudaFnConstWeighting.equiv(other)`
Test if `other` is an equivalent weighting.

Returnsequivalent : `bool`

`True` if `other` is a *FnWeightingBase* instance with the same *FnWeightingBase.impl*, which yields the same result as this inner product for any input, `False` otherwise. This is the same as equality if `other` is a *CudaFnConstWeighting* instance, otherwise by entry-wise comparison of this inner product's constant with the matrix of `other`.

CudaFnConstWeighting.inner
`CudaFnConstWeighting.inner(x1, x2)`
Calculate the constant-weighted inner product of two vectors.

Parametersx1, x2 : *CudaFnVector*

Vectors whose inner product is calculated

Returnsinner : `float` or `complex`

The inner product of the two vectors

CudaFnConstWeighting.norm
`CudaFnConstWeighting.norm(x)`
Calculate the constant-weighted norm of a vector.

Parametersx1 : *CudaFnVector*

Vector whose norm is calculated

Returnsnorm : `float`

The norm of the vector
__init__(*constant*, *exponent=2.0*)
Initialize a new instance.

Parametersconstant : positive finite `float`

Weighting constant of the inner product.

exponent : positive `float`

Exponent of the norm. For values other than 2.0, the inner product is not defined.

CudaFnCustomDist

class odl.space.cu_ntuples.**CudaFnCustomDist**(*dist*)

Bases: *odl.space.base_ntuples.FnWeightingBase*

Custom distance on *CudaFn*, removes `norm` and `inner`.

Attributes

exponent	Exponent of this weighting.
impl	Implementation backend of this weighting.

CudaFnCustomDist.exponent

CudaFnCustomDist.**exponent**

Exponent of this weighting.

CudaFnCustomDist.impl

CudaFnCustomDist.**impl**

Implementation backend of this weighting.

Methods

__eq__(other)	Return `self == other`.
dist(x1, x2)	Custom distance of this instance..
inner(x1, x2)	Inner product is not defined for custom distance.
norm(x)	Norm is not defined for custom distance.

CudaFnCustomDist.__eq__

CudaFnCustomDist.**__eq__**(*other*)

Return `self == other`.

Returnsequal : `bool`

`True` if `other` is a *CudaFnCustomDist* instance with the same norm, `False` otherwise.

CudaFnCustomDist.dist

CudaFnCustomDist.**dist**(*x1*, *x2*)

Custom distance of this instance..

CudaFnCustomDist.inner

CudaFnCustomDist.**inner**(*x1*, *x2*)

Inner product is not defined for custom distance.

CudaFnCustomDist.norm

CudaFnCustomDist.**norm**(*x*)

Norm is not defined for custom distance.

__init__(*dist*)
Initialize a new instance.

Parametersdist : `callable`

The distance function defining a metric on $\mathbb{F}^n$. It must accept two *CudaFnVector* arguments and fulfill the following mathematical conditions for any three vectors x, y, z:

- $d(x, y) = d(y, x)$
- $d(x, y) \geq 0$
- $d(x, y) = 0 \Leftrightarrow x = y$
- $d(x, y) \geq d(x, z) + d(z, y)$

CudaFnCustomInnerProduct

class `odl.space.cu_ntuples.`**CudaFnCustomInnerProduct**(*inner*, *dist_using_inner=True*)
Bases: *odl.space.base_ntuples.FnWeightingBase*

Custom inner product on *CudaFn*.

Attributes

exponent	Exponent of this weighting.
impl	Implementation backend of this weighting.

CudaFnCustomInnerProduct.exponent
`CudaFnCustomInnerProduct.`**exponent**
Exponent of this weighting.

CudaFnCustomInnerProduct.impl
`CudaFnCustomInnerProduct.`**impl**
Implementation backend of this weighting.

Methods

__eq__(other)	Return `self == other`.
inner(x1, x2)	Custom inner product of this instance..

CudaFnCustomInnerProduct.__eq__
`CudaFnCustomInnerProduct.`**__eq__**(*other*)
Return `self == other`.

Returnsequal : `bool`

`True` if `other` is a *CudaFnCustomInnerProduct* instance with the same inner product, `False` otherwise.

CudaFnCustomInnerProduct.inner

CudaFnCustomInnerProduct.**inner**(*x1*, *x2*)
Custom inner product of this instance..
__init__(*inner*, *dist_using_inner=True*)
Initialize a new instance.

Parametersinner : callable

The inner product implementation. It must accept two *CudaFnVector* arguments, return a complex number and satisfy the following conditions for all vectors x, y, z and scalars s:

- $\langle x, y\rangle = \overline{\langle y, x\rangle}$
- $\langle sx, y\rangle = s\langle x, y\rangle$
- $\langle x + z, y\rangle = \langle x, y\rangle + \langle z, y\rangle$
- $\langle x, x\rangle = 0 \Leftrightarrow x = 0$

dist_using_inner : bool, optional

Calculate dist using the formula

$\|x - y\|^2 = \|x\|^2 + \|y\|^2 - 2\Re\langle x, y\rangle$.

This avoids the creation of new arrays and is thus faster for large arrays. On the downside, it will not evaluate to exactly zero for equal (but not identical) x and y.

CudaFnCustomNorm

class odl.space.cu_ntuples.**CudaFnCustomNorm**(*norm*)
Bases: *odl.space.base_ntuples.FnWeightingBase*

Custom norm on *CudaFn*, removes inner.

Attributes

exponent	Exponent of this weighting.
impl	Implementation backend of this weighting.

CudaFnCustomNorm.exponent
CudaFnCustomNorm.**exponent**
Exponent of this weighting.

CudaFnCustomNorm.impl
CudaFnCustomNorm.**impl**
Implementation backend of this weighting.

Methods

__eq__(other)	Return self == other.
inner(x1, x2)	Inner product is not defined for custom distance.
norm(x)	Custom norm of this instance..

CudaFnCustomNorm.__eq__
`CudaFnCustomNorm.__eq__(other)`
Return `self == other`.

Returnsequal : `bool`

`True` if `other` is a *CudaFnCustomNorm* instance with the same norm, `False` otherwise.

CudaFnCustomNorm.inner
`CudaFnCustomNorm.inner(x1, x2)`
Inner product is not defined for custom distance.

CudaFnCustomNorm.norm
`CudaFnCustomNorm.norm(x)`
Custom norm of this instance..
`__init__(norm)`
Initialize a new instance.

Parametersnorm : `callable`

The norm implementation. It must accept an *CudaFnVector* argument, return a `float` and satisfy the following conditions for all vectors x, y and scalars s:

- $\|x\| \geq 0$
- $\|x\| = 0 \Leftrightarrow x = 0$
- $\|sx\| = |s|\|x\|$
- $\|x + y\| \leq \|x\| + \|y\|$.

CudaFnNoWeighting

class `odl.space.cu_ntuples.CudaFnNoWeighting(exponent=2.0)`
Bases: *odl.space.cu_ntuples.CudaFnConstWeighting*

Weighting of *CudaFn* with constant 1.

For exponent 2.0, the unweighted inner product is defined as

$\langle a, b \rangle := b^H a$

with b^H standing for transposed complex conjugate.

For other exponents, only norm and dist are defined.

Attributes

const	Weighting constant of this inner product.
exponent	Exponent of this weighting.
impl	Implementation backend of this weighting.

CudaFnNoWeighting.const
`CudaFnNoWeighting.const`
Weighting constant of this inner product.

CudaFnNoWeighting.exponent
CudaFnNoWeighting.**exponent**
Exponent of this weighting.

CudaFnNoWeighting.impl
CudaFnNoWeighting.**impl**
Implementation backend of this weighting.

Methods

__eq__(other)	Return `self == other`.
dist(x1, x2)	Calculate the constant-weighted distance between two vectors.
equiv(other)	Test if `other` is an equivalent weighting.
inner(x1, x2)	Calculate the constant-weighted inner product of two vectors.
norm(x)	Calculate the constant-weighted norm of a vector.

CudaFnNoWeighting.__eq__
CudaFnNoWeighting.**__eq__**(*other*)
Return `self == other`.

Returnsequal : `bool`

`True` if `other` is a *CudaFnConstWeighting* instance with the same constant, `False` otherwise.

CudaFnNoWeighting.dist
CudaFnNoWeighting.**dist**(*x1*, *x2*)
Calculate the constant-weighted distance between two vectors.

Parametersx1, x2 : *CudaFnVector*

Vectors whose mutual distance is calculated

Returnsdist : `float`

The distance between the vectors

CudaFnNoWeighting.equiv
CudaFnNoWeighting.**equiv**(*other*)
Test if `other` is an equivalent weighting.

Returnsequivalent : `bool`

`True` if `other` is a *FnWeightingBase* instance with the same *FnWeightingBase.impl*, which yields the same result as this inner product for any input, `False` otherwise. This is the same as equality if `other` is a *CudaFnConstWeighting* instance, otherwise by entry-wise comparison of this inner product's constant with the matrix of `other`.

CudaFnNoWeighting.inner
CudaFnNoWeighting.**inner**(*x1*, *x2*)
Calculate the constant-weighted inner product of two vectors.

Parametersx1, x2 : *CudaFnVector*

Vectors whose inner product is calculated

Returnsinner : `float` or `complex`

The inner product of the two vectors

CudaFnNoWeighting.norm

`CudaFnNoWeighting.`**`norm`**(*x*)

Calculate the constant-weighted norm of a vector.

Parametersx1 : *CudaFnVector*

Vector whose norm is calculated

Returnsnorm : `float`

The norm of the vector

`__init__`(*exponent=2.0*)

Initialize a new instance.

CudaFnVector

class `odl.space.cu_ntuples.`**`CudaFnVector`**(*space*, *data*)

Bases: *odl.space.base_ntuples.FnBaseVector*, *odl.space.cu_ntuples.CudaNtuplesVector*

Representation of a *CudaFn* element.

Attributes

T	The transpose of a vector, the functional given by (.
data	The data of this vector.
data_ptr	A raw pointer to the data of this vector.
dtype	Length of this vector, equal to space size.
itemsize	The size in bytes on one element of this type.
nbytes	The number of bytes this vector uses in memory.
ndim	Number of dimensions, always 1.
shape	Shape of this vector, equals `(size,)`.
size	Length of this vector, equal to space size.
space	Space to which this vector.
ufunc	*CudaNtuplesUFuncs*, access to numpy style ufuncs.

CudaFnVector.T

`CudaFnVector.`**`T`**

The transpose of a vector, the functional given by (. , self)

Returnstranspose : *InnerProductOperator*

Notes

This function is only defined in inner product spaces.

In a complex space, this takes the conjugate transpose of the vector.

Examples

```
>>> from odl import Rn
>>> import numpy as np
>>> rn = Rn(3)
>>> x = rn.element([1, 2, 3])
>>> y = rn.element([2, 1, 3])
>>> x.T(y)
13.0
```

CudaFnVector.data

CudaFnVector.**data**

The data of this vector.

ParametersNone

Returnsptr : CudaFnVectorImpl

Underlying cuda data representation

CudaFnVector.data_ptr

CudaFnVector.**data_ptr**

A raw pointer to the data of this vector.

CudaFnVector.dtype

CudaFnVector.**dtype**

Length of this vector, equal to space size.

CudaFnVector.itemsize

CudaFnVector.**itemsize**

The size in bytes on one element of this type.

CudaFnVector.nbytes

CudaFnVector.**nbytes**

The number of bytes this vector uses in memory.

CudaFnVector.ndim

CudaFnVector.**ndim**

Number of dimensions, always 1.

CudaFnVector.shape

CudaFnVector.**shape**

Shape of this vector, equals `(size,)`.

CudaFnVector.size

CudaFnVector.**size**

Length of this vector, equal to space size.

CudaFnVector.space

CudaFnVector.**space**

Space to which this vector.

CudaFnVector.ufunc

CudaFnVector.**ufunc**

CudaNtuplesUFuncs, access to numpy style ufuncs.

See also:

NtuplesBaseUFuncsBase class for ufuncs in *NtuplesBase* spaces.

Notes

Not all ufuncs are currently optimized, some use the default numpy implementation. This can be improved in the future.

Examples

```
>>> r2 = CudaRn(2)
>>> x = r2.element([1, -2])
>>> x.ufunc.absolute()
CudaRn(2).element([1.0, 2.0])
```

These functions can also be used with broadcasting

```
>>> x.ufunc.add(3)
CudaRn(2).element([4.0, 1.0])
```

and non-space elements

```
>>> x.ufunc.subtract([3, 3])
CudaRn(2).element([-2.0, -5.0])
```

There is also support for various reductions (sum, prod, min, max)

```
>>> x.ufunc.sum()
-1.0
```

Also supports out parameter

```
>>> y = r2.element([3, 4])
>>> out = r2.element()
>>> result = x.ufunc.add(y, out=out)
>>> result
CudaRn(2).element([4.0, 2.0])
>>> result is out
True
```

Methods

__getitem__(indices)	Access values of this vector.
__setitem__(indices, values)	Set values of this vector.

Continued on next page

Table 2.168 – continued from previous page

asarray([start, stop, step, out])	Extract the data of this array as a numpy array.

CudaFnVector.__getitem__

`CudaFnVector.__getitem__`(*indices*)

Access values of this vector.

This will cause the values to be copied to CPU which is a slow operation.

Parametersindices : `int` or `slice`

The position(s) that should be accessed

Returnsvalues : scalar or *CudaNtuplesVector*

The value(s) at the index (indices)

Examples

```
>>> uc3 = CudaNtuples(3, 'uint8')
>>> y = uc3.element([1, 2, 3])
>>> y[0]
1
>>> z = y[1:3]
>>> z
CudaNtuples(2, 'uint8').element([2, 3])
>>> y[::2]
CudaNtuples(2, 'uint8').element([1, 3])
>>> y[::-1]
CudaNtuples(3, 'uint8').element([3, 2, 1])
```

The returned value is a view, modifications are reflected in the original data:

```
>>> z[:] = [4, 5]
>>> y
CudaNtuples(3, 'uint8').element([1, 4, 5])
```

CudaFnVector.__setitem__

`CudaFnVector.__setitem__`(*indices*, *values*)

Set values of this vector.

This will cause the values to be copied to CPU which is a slow operation.

Parametersindices : `int` or `slice`

The position(s) that should be set

values : {scalar, array-like, *CudaNtuplesVector*}

The value(s) that are to be assigned.

If `index` is an `int`, `value` must be single value.

If `index` is a `slice`, `value` must be broadcastable to the size of the slice (same size, shape (1,) or single value).

Returns`None`

Examples

```
>>> uc3 = CudaNtuples(3, 'uint8')
>>> y = uc3.element([1, 2, 3])
>>> y[0] = 5
>>> y
CudaNtuples(3, 'uint8').element([5, 2, 3])
>>> y[1:3] = [7, 8]
>>> y
CudaNtuples(3, 'uint8').element([5, 7, 8])
>>> y[:] = np.array([0, 0, 0])
>>> y
CudaNtuples(3, 'uint8').element([0, 0, 0])
```

Scalar assignment

```
>>> y[:] = 5
>>> y
CudaNtuples(3, 'uint8').element([5, 5, 5])
```

CudaFnVector.asarray

CudaFnVector.**asarray**(*start=None*, *stop=None*, *step=None*, *out=None*)

Extract the data of this array as a numpy array.

Parametersstart : int, optional

Start position. None means the first element.

start : int, optional

One element past the last element to be extracted. None means the last element.

start : int, optional

Step length. None means 1.

out : numpy.ndarray

Array in which the result should be written in-place. Has to be contiguous and of the correct dtype.

Returnsasarray : numpy.ndarray

Numpy array of the same type as the space.

Examples

```
>>> uc3 = CudaNtuples(3, 'uint8')
>>> y = uc3.element([1, 2, 3])
>>> y.asarray()
array([1, 2, 3], dtype=uint8)
>>> y.asarray(1, 3)
array([2, 3], dtype=uint8)
```

Using the out parameter

```
>>> out = np.empty((3,), dtype='uint8')
>>> result = y.asarray(out=out)
>>> out
```

```
        array([1, 2, 3], dtype=uint8)
        >>> result is out
        True
```

__init__ (*space*, *data*)
Initialize a new instance.

CudaFnVectorWeighting

class odl.space.cu_ntuples.**CudaFnVectorWeighting** (*vector*, *exponent=2.0*)
Bases: *odl.space.base_ntuples.FnWeightingBase*

Vector weighting for *CudaFn*.

For exponent 2.0, a new weighted inner product with vector w is defined as

$\langle a, b \rangle_w := b^H (w \odot a)$

with b^H standing for transposed complex conjugate, and $w \odot a$ being element-wise multiplication.

For other exponents, only norm and dist are defined. In the case of exponent `inf`, the weighted norm is

$\|a\|_{w,\infty} := \|w \odot a\|_\infty$,

otherwise it is

$\|a\|_{w,p} := \|w^{1/p} \odot a\|_p$.

Not that this definition does **not** fulfill the limit property in p, i.e.

$\lim_{p \to \infty} \|a\|_{w,p} = \|a\|_\infty \neq \|a\|_{w,\infty}$

unless $w = (1, \ldots, 1)$.

The vector may only have positive entries, otherwise it does not define an inner product or norm, respectively. This is not checked during initialization.

Attributes

exponent	Exponent of this weighting.
impl	Implementation backend of this weighting.
vector	Weighting vector of this inner product.

CudaFnVectorWeighting.exponent
CudaFnVectorWeighting.**exponent**
Exponent of this weighting.

CudaFnVectorWeighting.impl
CudaFnVectorWeighting.**impl**
Implementation backend of this weighting.

CudaFnVectorWeighting.vector
CudaFnVectorWeighting.**vector**
Weighting vector of this inner product.

Methods

__eq__(other)	Return `self == other`.
dist(x1, x2)	Calculate the vector-weighted distance between two vectors.
equiv(other)	Test if `other` is an equivalent weighting.
inner(x1, x2)	Calculate the vector weighted inner product of two vectors.
norm(x)	Calculate the vector-weighted norm of a vector.
vector_is_valid()	Test if the vector is a valid weight, i.e.

CudaFnVectorWeighting.__eq__

`CudaFnVectorWeighting.`**`__eq__`**(*other*)

Return `self == other`.

Returnsequals : `bool`

`True` if `other` is a *CudaFnVectorWeighting* instance with **identical** vector, `False` otherwise.

See also:

equivtest for equivalent inner products

CudaFnVectorWeighting.dist

`CudaFnVectorWeighting.`**`dist`**(*x1*, *x2*)

Calculate the vector-weighted distance between two vectors.

Parametersx1, x2 : *CudaFnVector*

Vectors whose mutual distance is calculated

Returnsdist : `float`

The distance between the vectors

CudaFnVectorWeighting.equiv

`CudaFnVectorWeighting.`**`equiv`**(*other*)

Test if `other` is an equivalent weighting.

Returnsequivalent : `bool`

`True` if `other` is a *FnWeightingBase* instance which yields the same result as this inner product for any input, `False` otherwise. This is checked by entry-wise comparison of matrices/vectors/constant of this inner product and `other`.

CudaFnVectorWeighting.inner

`CudaFnVectorWeighting.`**`inner`**(*x1*, *x2*)

Calculate the vector weighted inner product of two vectors.

Parametersx1, x2 : *CudaFnVector*

Vectors whose inner product is calculated

Returnsinner : `float` or `complex`

The inner product of the two provided vectors

CudaFnVectorWeighting.norm
CudaFnVectorWeighting.**norm**(*x*)
Calculate the vector-weighted norm of a vector.

Parametersx : *CudaFnVector*

Vector whose norm is calculated

Returnsnorm : float

The norm of the provided vector

CudaFnVectorWeighting.vector_is_valid
CudaFnVectorWeighting.**vector_is_valid**()
Test if the vector is a valid weight, i.e. positive.

Notes

This operation copies the vector to the CPU memory if necessary and uses numpy.all, which can be very time-consuming in total.

__init__(*vector*, *exponent=2.0*)
Initialize a new instance.

Parametersvector : array-like, one-dim.

Weighting vector of the inner product

exponent : positive float

Exponent of the norm. For values other than 2.0, the inner product is not defined.

CudaNtuples

class odl.space.cu_ntuples.**CudaNtuples**(*size*, *dtype*)
Bases: *odl.space.base_ntuples.NtuplesBase*

The space *NtuplesBase*, implemented in CUDA.

Attributes

dtype	The data type of each entry.
element_type	*CudaNtuplesVector*
size	The number of entries per tuple.

CudaNtuples.dtype
CudaNtuples.**dtype**
The data type of each entry.

CudaNtuples.element_type
CudaNtuples.**element_type**
CudaNtuplesVector

CudaNtuples.size

CudaNtuples.**size**
The number of entries per tuple.

Methods

element([inp, data_ptr])	Create a new element.

CudaNtuples.element
CudaNtuples.**element**(*inp=None*, *data_ptr=None*)
Create a new element.

Parametersinp : array-like or scalar, optional

Input to initialize the new element.

If `inp` is a `numpy.ndarray` of shape `(size,)` and the same data type as this space, the array is wrapped, not copied. Other array-like objects are copied (with broadcasting if necessary).

If a single value is given, it is copied to all entries.

If both `inp` and `data_ptr` are `None`, an empty element is created with no guarantee of its state (memory allocation only).

data_ptr : `int`, optional

Memory address of a CUDA array container

Cannot be combined with `inp`.

Returnselement : *CudaNtuplesVector*

The new element

Notes

This method preserves "array views" of correct size and type, see the examples below.

TODO: No, it does not yet!

Examples

```
>>> uc3 = CudaNtuples(3, 'uint8')
>>> x = uc3.element(np.array([1, 2, 3], dtype='uint8'))
>>> x
CudaNtuples(3, 'uint8').element([1, 2, 3])
>>> y = uc3.element([1, 2, 3])
>>> y
CudaNtuples(3, 'uint8').element([1, 2, 3])
```

__init__(*size*, *dtype*)
Initialize a new instance.

Parameterssize : `int`

The number entries per tuple

dtype : `object`

The data type for each tuple entry. Can be provided in any way the `numpy.dtype` function understands, most notably as built-in type, as one of NumPy's internal datatype objects or as string.

Check `CUDA_DTYPES` for a list of available data types.

CudaNtuplesVector

class `odl.space.cu_ntuples.`**CudaNtuplesVector**(*space*, *data*)

Bases: *odl.space.base_ntuples.NtuplesBaseVector*, *odl.set.space.LinearSpaceVector*

Representation of a *CudaNtuples* element.

Attributes

T	The transpose of a vector, the functional given by (.
data	The data of this vector.
data_ptr	A raw pointer to the data of this vector.
dtype	Length of this vector, equal to space size.
itemsize	The size in bytes on one element of this type.
nbytes	The number of bytes this vector uses in memory.
ndim	Number of dimensions, always 1.
shape	Shape of this vector, equals `(size,)`.
size	Length of this vector, equal to space size.
space	Space to which this vector.
ufunc	*CudaNtuplesUFuncs*, access to numpy style ufuncs.

CudaNtuplesVector.T

`CudaNtuplesVector.`**T**

The transpose of a vector, the functional given by (. , self)

Returnstranspose : *InnerProductOperator*

Notes

This function is only defined in inner product spaces.

In a complex space, this takes the conjugate transpose of the vector.

Examples

```
>>> from odl import Rn
>>> import numpy as np
>>> rn = Rn(3)
>>> x = rn.element([1, 2, 3])
>>> y = rn.element([2, 1, 3])
>>> x.T(y)
13.0
```

CudaNtuplesVector.data

CudaNtuplesVector.**data**
The data of this vector.

ParametersNone

Returnsptr : CudaFnVectorImpl

Underlying cuda data representation

CudaNtuplesVector.data_ptr
CudaNtuplesVector.**data_ptr**
A raw pointer to the data of this vector.

CudaNtuplesVector.dtype
CudaNtuplesVector.**dtype**
Length of this vector, equal to space size.

CudaNtuplesVector.itemsize
CudaNtuplesVector.**itemsize**
The size in bytes on one element of this type.

CudaNtuplesVector.nbytes
CudaNtuplesVector.**nbytes**
The number of bytes this vector uses in memory.

CudaNtuplesVector.ndim
CudaNtuplesVector.**ndim**
Number of dimensions, always 1.

CudaNtuplesVector.shape
CudaNtuplesVector.**shape**
Shape of this vector, equals `(size,)`.

CudaNtuplesVector.size
CudaNtuplesVector.**size**
Length of this vector, equal to space size.

CudaNtuplesVector.space
CudaNtuplesVector.**space**
Space to which this vector.

CudaNtuplesVector.ufunc
CudaNtuplesVector.**ufunc**
CudaNtuplesUFuncs, access to numpy style ufuncs.

See also:

NtuplesBaseUFuncsBase class for ufuncs in *NtuplesBase* spaces.

Notes

Not all ufuncs are currently optimized, some use the default numpy implementation. This can be improved in the future.

Examples

```
>>> r2 = CudaRn(2)
>>> x = r2.element([1, -2])
>>> x.ufunc.absolute()
CudaRn(2).element([1.0, 2.0])
```

These functions can also be used with broadcasting

```
>>> x.ufunc.add(3)
CudaRn(2).element([4.0, 1.0])
```

and non-space elements

```
>>> x.ufunc.subtract([3, 3])
CudaRn(2).element([-2.0, -5.0])
```

There is also support for various reductions (sum, prod, min, max)

```
>>> x.ufunc.sum()
-1.0
```

Also supports out parameter

```
>>> y = r2.element([3, 4])
>>> out = r2.element()
>>> result = x.ufunc.add(y, out=out)
>>> result
CudaRn(2).element([4.0, 2.0])
>>> result is out
True
```

Methods

__eq__(other)	Return `self == other`.
__getitem__(indices)	Access values of this vector.
__setitem__(indices, values)	Set values of this vector.
asarray([start, stop, step, out])	Extract the data of this array as a numpy array.
copy()	Create an identical (deep) copy of this vector.

CudaNtuplesVector.__eq__

`CudaNtuplesVector.__eq__`(*other*)

Return `self == other`.

Returnsequals : `bool`

`True` if all elements of `other` are equal to this vector's elements, `False` otherwise

Examples

```
>>> r3 = CudaNtuples(3, 'float32')
>>> x = r3.element([1, 2, 3])
>>> x == x
True
>>> y = r3.element([1, 2, 3])
>>> x == y
True
>>> y = r3.element([0, 0, 0])
>>> x == y
False
>>> r3_2 = CudaNtuples(3, 'uint8')
>>> z = r3_2.element([1, 2, 3])
>>> x != z
True
```

CudaNtuplesVector.__getitem__

`CudaNtuplesVector.__getitem__`(*indices*)

Access values of this vector.

This will cause the values to be copied to CPU which is a slow operation.

Parametersindices : `int` or `slice`

The position(s) that should be accessed

Returnsvalues : scalar or *CudaNtuplesVector*

The value(s) at the index (indices)

Examples

```
>>> uc3 = CudaNtuples(3, 'uint8')
>>> y = uc3.element([1, 2, 3])
>>> y[0]
1
>>> z = y[1:3]
>>> z
CudaNtuples(2, 'uint8').element([2, 3])
>>> y[::2]
CudaNtuples(2, 'uint8').element([1, 3])
>>> y[::-1]
CudaNtuples(3, 'uint8').element([3, 2, 1])
```

The returned value is a view, modifications are reflected in the original data:

```
>>> z[:] = [4, 5]
>>> y
CudaNtuples(3, 'uint8').element([1, 4, 5])
```

CudaNtuplesVector.__setitem__

`CudaNtuplesVector.__setitem__`(*indices*, *values*)

Set values of this vector.

This will cause the values to be copied to CPU which is a slow operation.

Parametersindices : `int` or `slice`

The position(s) that should be set

values : {scalar, array-like, *`CudaNtuplesVector`*}

The value(s) that are to be assigned.

If `index` is an `int`, `value` must be single value.

If `index` is a `slice`, `value` must be broadcastable to the size of the slice (same size, shape (1,) or single value).

Returns`None`

Examples

```
>>> uc3 = CudaNtuples(3, 'uint8')
>>> y = uc3.element([1, 2, 3])
>>> y[0] = 5
>>> y
CudaNtuples(3, 'uint8').element([5, 2, 3])
>>> y[1:3] = [7, 8]
>>> y
CudaNtuples(3, 'uint8').element([5, 7, 8])
>>> y[:] = np.array([0, 0, 0])
>>> y
CudaNtuples(3, 'uint8').element([0, 0, 0])
```

Scalar assignment

```
>>> y[:] = 5
>>> y
CudaNtuples(3, 'uint8').element([5, 5, 5])
```

CudaNtuplesVector.asarray

`CudaNtuplesVector.`**`asarray`**(*start=None*, *stop=None*, *step=None*, *out=None*)

Extract the data of this array as a numpy array.

Parametersstart : `int`, optional

Start position. None means the first element.

start : `int`, optional

One element past the last element to be extracted. None means the last element.

start : `int`, optional

Step length. None means 1.

out : `numpy.ndarray`

Array in which the result should be written in-place. Has to be contiguous and of the correct dtype.

Returnsasarray : `numpy.ndarray`

Numpy array of the same type as the space.

Examples

```
>>> uc3 = CudaNtuples(3, 'uint8')
>>> y = uc3.element([1, 2, 3])
>>> y.asarray()
array([1, 2, 3], dtype=uint8)
>>> y.asarray(1, 3)
array([2, 3], dtype=uint8)
```

Using the out parameter

```
>>> out = np.empty((3,), dtype='uint8')
>>> result = y.asarray(out=out)
>>> out
array([1, 2, 3], dtype=uint8)
>>> result is out
True
```

CudaNtuplesVector.copy

CudaNtuplesVector.**copy**()

Create an identical (deep) copy of this vector.

Returnscopy : *CudaNtuplesVector*

The deep copy

Examples

```
>>> vec1 = CudaNtuples(3, 'uint8').element([1, 2, 3])
>>> vec2 = vec1.copy()
>>> vec2
CudaNtuples(3, 'uint8').element([1, 2, 3])
>>> vec1 == vec2
True
>>> vec1 is vec2
False
```

__init__(*space*, *data*)

Initialize a new instance.

Functions

CudaRn(size[, dtype])	The real space R^n, implemented in CUDA.
cu_weighted_dist(weight[, exponent])	Weighted distance on *CudaFn* spaces as free function.
cu_weighted_inner(weight)	Weighted inner product on *CudaFn* spaces as free function.
cu_weighted_norm(weight[, exponent])	Weighted norm on *CudaFn* spaces as free function.

CudaRn

odl.space.cu_ntuples.**CudaRn**(*size*, *dtype=<Mock id='140323975437112'>*, ***kwargs*)

The real space R^n, implemented in CUDA.

Requires the compiled ODL extension odlpp.

Parameterssize : positive `int`

The number of dimensions of the space

dtype : `object`

The data type of the storage array. Can be provided in any way the `numpy.dtype` function understands, most notably as built-in type, as one of NumPy's internal datatype objects or as string.

Only real floating-point data types are allowed.

kwargs : {'weight', 'exponent', 'dist', 'norm', 'inner'}

See *CudaFn*

cu_weighted_dist

odl.space.cu_ntuples.**cu_weighted_dist**(*weight*, *exponent=2.0*)

Weighted distance on *CudaFn* spaces as free function.

Parametersweight : scalar, array-like or *CudaFnVector*

Weight of the inner product. A scalar is interpreted as a constant weight and a 1-dim. array or a *CudaFnVector* as a weighting vector.

exponent : positive `float`

Exponent of the distance

Returnsdist : `callable`

Distance function with given weight. Constant weightings are applicable to spaces of any size, for arrays the sizes of the weighting and the space must match.

See also:

CudaFnConstWeighting, *CudaFnVectorWeighting*

cu_weighted_inner

odl.space.cu_ntuples.**cu_weighted_inner**(*weight*)

Weighted inner product on *CudaFn* spaces as free function.

Parametersweight : scalar, array-like or *CudaFnVector*

Weight of the inner product. A scalar is interpreted as a constant weight and a 1-dim. array or a *CudaFnVector* as a weighting vector.

Returnsinner : `callable`

Inner product function with given weight. Constant weightings are applicable to spaces of any size, for arrays the sizes of the weighting and the space must match.

See also:

CudaFnConstWeighting, *CudaFnVectorWeighting*

cu_weighted_norm

odl.space.cu_ntuples.**cu_weighted_norm**(*weight*, *exponent=2.0*)

Weighted norm on *CudaFn* spaces as free function.

Parametersweight : scalar, array-like or *CudaFnVector*

Weight of the inner product. A scalar is interpreted as a constant weight and a 1-dim. array or a *CudaFnVector* as a weighting vector.

exponent : positive `float`

Exponent of the norm. If `weight` is a sparse matrix, only 1.0, 2.0 and `inf` are allowed.

Returnsnorm : `callable`

Norm function with given weight. Constant weightings are applicable to spaces of any size, for arrays the sizes of the weighting and the space must match.

See also:

CudaFnConstWeighting, *CudaFnVectorWeighting*

fspace

Spaces of functions with common domain and range.

Classes

FunctionSet(dom, ran)	A general set of functions with common domain and range.
FunctionSetVector(fset[, fcall, fapply])	Representation of a *FunctionSet* element.
FunctionSpace(dom[, field])	A vector space of functions.
FunctionSpaceVector(fspace[, fcall, fapply])	Representation of a *FunctionSpace* element.

FunctionSet

class odl.space.fspace.**FunctionSet**(*dom*, *ran*)

Bases: *odl.set.sets.Set*

A general set of functions with common domain and range.

Attributes

domain	Return domain attribute.
element_type	*FunctionSetVector*
range	Return range attribute.

FunctionSet.domain

FunctionSet.**domain**

Return domain attribute.

FunctionSet.element_type

FunctionSet.**element_type**
FunctionSetVector

FunctionSet.range
FunctionSet.**range**
Return range attribute.

Methods

__contains__(other)	Return `other in self`.
__eq__(other)	Return `self == other`.
element([fcall, fapply])	Create a *FunctionSet* element.

FunctionSet.__contains__
FunctionSet.**__contains__**(*other*)
Return `other in self`.

Returnsequals : `bool`

`True` if `other` is a *FunctionSetVector* whose *FunctionSetVector.space* attribute equals this space, `False` otherwise.

FunctionSet.__eq__
FunctionSet.**__eq__**(*other*)
Return `self == other`.

Returnsequals : `bool`

`True` if `other` is a *FunctionSet* with same *FunctionSet.domain* and *FunctionSet.range*, `False` otherwise.

FunctionSet.element
FunctionSet.**element**(*fcall=None*, *fapply=None*)
Create a *FunctionSet* element.

Parametersfcall : `callable`, optional

The actual instruction for out-of-place evaluation. It must return an *range* element or a `numpy.ndarray` of such (vectorized call).

If fcall is a *FunctionSetVector*, it is wrapped as a new *FunctionSetVector*.

fapply : `callable`, optional

The actual instruction for in-place evaluation. Its first argument must be the *range* element or the array of such (vectorization) to which the result is written.

If fapply‘ is a *FunctionSetVector*, it is wrapped as a new *FunctionSetVector*.

***At least one of the arguments fcall and fapply must**

be provided.*

Returnselement : *FunctionSetVector*

The new element created

__init__ (*dom*, *ran*)
Initialize a new instance.

Parametersdom : *Set*

The domain of the functions.

ran : *Set*

The range of the functions.

FunctionSetVector

class odl.space.fspace.**FunctionSetVector** (*fset*, *fcall=None*, *fapply=None*)
Bases: *odl.operator.operator.Operator*

Representation of a *FunctionSet* element.

Attributes

adjoint	The operator adjoint (abstract).
domain	The domain of this operator.
inverse	Return the operator inverse.
is_functional	`True` if the this operator's range is a *Field*.
is_linear	`True` if this operator is linear.
range	The range of this operator.
space	Return space attribute.

FunctionSetVector.adjoint
FunctionSetVector.**adjoint**
The operator adjoint (abstract).

RaisesOpNotImplementedError

Since the adjoint cannot be default implemented.

FunctionSetVector.domain
FunctionSetVector.**domain**
The domain of this operator.

FunctionSetVector.inverse
FunctionSetVector.**inverse**
Return the operator inverse.

RaisesOpNotImplementedError

Since the inverse cannot be default implemented.

FunctionSetVector.is_functional
FunctionSetVector.**is_functional**
`True` if the this operator's range is a *Field*.

FunctionSetVector.is_linear
`FunctionSetVector.is_linear`
`True` if this operator is linear.

FunctionSetVector.range
`FunctionSetVector.range`
The range of this operator.

FunctionSetVector.space
`FunctionSetVector.space`
Return space attribute.

Methods

__call__(*x)	Vectorized and multi-argument out-of-place evaluation.
__eq__(other)	Return `self == other`.
apply(out, *x)	Vectorized and multi-argument in-place evaluation.

FunctionSetVector.__call__
`FunctionSetVector.__call__`(**x*)
Vectorized and multi-argument out-of-place evaluation.

Parametersx1,...,xN : `object`

Input arguments for the function evaluation.

Returnsout : *FunctionSet.range* element or array of elements

Result of the function evaluation.

FunctionSetVector.__eq__
`FunctionSetVector.__eq__`(*other*)
Return `self == other`.

Returnsequals : `bool`

`True` if `other` is a *FunctionSetVector* with `other.space` equal to this vector's space and the call and apply implementations of `other` and this vector are equal. `False` otherwise.

FunctionSetVector.apply
`FunctionSetVector.apply`(*out*, **x*)
Vectorized and multi-argument in-place evaluation.

Parametersout : *FunctionSet.range* element or array of elements

Element(s) to which the result is written.

inp1,...,inpN : `object`

Input arguments for the function evaluation.

Returns`None`
__init__(*fset*, *fcall=None*, *fapply=None*)
Initialize a new instance.

Parametersfset : *FunctionSet*

The set of functions this element lives in

fcall : `callable`, optional

The actual instruction for out-of-place evaluation. It must return an *FunctionSet.range* element or a `numpy.ndarray` of such (vectorized call).

fapply : `callable`, optional

The actual instruction for in-place evaluation. Its first argument must be the *FunctionSet.range* element or the array of such (vectorization) to which the result is written.

***At least one of the arguments fcall and fapply‘ must**

be provided.*

FunctionSpace

class `odl.space.fspace.`**FunctionSpace**(*dom*, *field=RealNumbers()*)

Bases: *odl.space.fspace.FunctionSet*, *odl.set.space.LinearSpace*

A vector space of functions.

Attributes

domain	Return domain attribute.
element_type	*FunctionSpaceVector*
field	The field of this vector space.
range	Return range attribute.

FunctionSpace.domain

`FunctionSpace.`**domain**

Return domain attribute.

FunctionSpace.element_type

`FunctionSpace.`**element_type**

FunctionSpaceVector

FunctionSpace.field

`FunctionSpace.`**field**

The field of this vector space.

The field is the set of scalars of the space, that is numbers that the vectors in the space can be multiplied with.

Returnsfield : *Field*

The underlying field.

FunctionSpace.range

FunctionSpace.**range**
Return range attribute.

Methods

__contains__(other)	Return `other in self`.
__eq__(other)	Return `self == other`.
_lincomb(a, x, b, y, out)	Raw linear combination of `x` and `y`.
_multiply(x, y)	Raw pointwise multiplication of two functions.
element([fcall, fapply])	Create a *FunctionSpace* element.
zero()	The function mapping everything to zero.

FunctionSpace.__contains__
FunctionSpace.**__contains__**(*other*)
Return `other in self`.

Returnsequals : `bool`

`True` if `other` is a *FunctionSetVector* whose *FunctionSetVector.space* attribute equals this space, `False` otherwise.

FunctionSpace.__eq__
FunctionSpace.**__eq__**(*other*)
Return `self == other`.

Returnsequals : `bool`

`True` if `other` is a *FunctionSpace* with same *FunctionSet.domain* and *FunctionSet.range*, `False` otherwise.

FunctionSpace._lincomb
FunctionSpace.**_lincomb**(*a*, *x*, *b*, *y*, *out*)
Raw linear combination of `x` and `y`.

Notes

The additions and multiplications are implemented via a simple Python function, so the resulting function is probably slow.

FunctionSpace._multiply
FunctionSpace.**_multiply**(*x*, *y*)
Raw pointwise multiplication of two functions.

Notes

The multiplication is implemented with a simple Python function, so the resulting function object is probably slow.

FunctionSpace.element
`FunctionSpace.`**`element`**(*fcall=None*, *fapply=None*)
Create a *FunctionSpace* element.

Parametersfcall : `callable`, optional

The actual instruction for out-of-place evaluation. It must return an *FunctionSet.range* element or a `numpy.ndarray` of such (vectorized call).

If fcall is a *FunctionSetVector*, it is wrapped as a new *FunctionSpaceVector*.

fapply : `callable`, optional

The actual instruction for in-place evaluation. Its first argument must be the *FunctionSet.range* element or the array of such (vectorization) to which the result is written.

If fapply is a *FunctionSetVector*, it is wrapped as a new *FunctionSpaceVector*.

Returnselement : *FunctionSpaceVector*

The new element.

FunctionSpace.zero
`FunctionSpace.`**`zero`**()
The function mapping everything to zero.

Notes

Since *FunctionSpace._lincomb* is slow, we implement this function directly.
`__init__`(*dom*, *field=RealNumbers()*)
Initialize a new instance.

Parametersdom : *Set*

The domain of the functions.

field : *RealNumbers* or *ComplexNumbers*

The range of the functions.

FunctionSpaceVector

class `odl.space.fspace.`**`FunctionSpaceVector`**(*fspace*, *fcall=None*, *fapply=None*)
Bases: *odl.space.fspace.FunctionSetVector*, *odl.set.space.LinearSpaceVector*

Representation of a *FunctionSpace* element.

Attributes

T	The transpose of a vector, the functional given by (.
adjoint	The operator adjoint (abstract).

Continued on next page

Table 2.183 – continued from previous page

domain	The domain of this operator.
inverse	Return the operator inverse.
is_functional	`True` if the this operator's range is a *Field*.
is_linear	`True` if this operator is linear.
range	The range of this operator.
space	Return space attribute.

FunctionSpaceVector.T

`FunctionSpaceVector.T`

The transpose of a vector, the functional given by (. , self)

Returnstranspose : *InnerProductOperator*

Notes

This function is only defined in inner product spaces.

In a complex space, this takes the conjugate transpose of the vector.

Examples

```
>>> from odl import Rn
>>> import numpy as np
>>> rn = Rn(3)
>>> x = rn.element([1, 2, 3])
>>> y = rn.element([2, 1, 3])
>>> x.T(y)
13.0
```

FunctionSpaceVector.adjoint

`FunctionSpaceVector.adjoint`

The operator adjoint (abstract).

RaisesOpNotImplementedError

Since the adjoint cannot be default implemented.

FunctionSpaceVector.domain

`FunctionSpaceVector.domain`

The domain of this operator.

FunctionSpaceVector.inverse

`FunctionSpaceVector.inverse`

Return the operator inverse.

RaisesOpNotImplementedError

Since the inverse cannot be default implemented.

FunctionSpaceVector.is_functional
`FunctionSpaceVector.`**`is_functional`**
`True` if the this operator's range is a *`Field`*.

FunctionSpaceVector.is_linear
`FunctionSpaceVector.`**`is_linear`**
`True` if this operator is linear.

FunctionSpaceVector.range
`FunctionSpaceVector.`**`range`**
The range of this operator.

FunctionSpaceVector.space
`FunctionSpaceVector.`**`space`**
Return space attribute.

Methods

`__call__`(*x)	Vectorized and multi-argument out-of-place evaluation.
`__eq__`(other)	Return `self == other`.
`apply`(out, *x)	Vectorized and multi-argument in-place evaluation.

FunctionSpaceVector.__call__
`FunctionSpaceVector.`**`__call__`**(**x*)
Vectorized and multi-argument out-of-place evaluation.

Parametersx1,...,xN : `object`

Input arguments for the function evaluation.

Returnsout : *`FunctionSet.range`* element or array of elements

Result of the function evaluation.

FunctionSpaceVector.__eq__
`FunctionSpaceVector.`**`__eq__`**(*other*)
Return `self == other`.

Returnsequals : `bool`

`True` if `other` is a *`FunctionSetVector`* with `other.space` equal to this vector's space and the call and apply implementations of `other` and this vector are equal. `False` otherwise.

FunctionSpaceVector.apply
`FunctionSpaceVector.`**`apply`**(*out*, **x*)
Vectorized and multi-argument in-place evaluation.

Parametersout : *`FunctionSet.range`* element or array of elements

Element(s) to which the result is written.

inp1,...,inpN : `object`

Input arguments for the function evaluation.

Returns None

__init__ (*fspace*, *fcall=None*, *fapply=None*)

Initialize a new instance.

Parametersfspace : *FunctionSpace*

The set of functions this element lives in

fcall : callable, optional

The actual instruction for out-of-place evaluation. It must return an *FunctionSet.range* element or a numpy.ndarray of such (vectorized call).

fapply : callable, optional

The actual instruction for in-place evaluation. Its first argument must be the *FunctionSet.range* element or the array of such (vectorization) to which the result is written.

***At least one of the arguments ‘‘fcall‘‘ and ‘‘fapply‘‘ must**

be provided.*

ntuples

CPU implementations of n-dimensional Cartesian spaces.

This is a default implementation of A^n for an arbitrary set A as well as the real and complex spaces R^n and C^n. The data is represented by NumPy arrays.

Classes

Fn(size, dtype, **kwargs)	The vector space $\mathbb{F}^n$ with vector multiplication.
FnConstWeighting(constant[, exponent, ...])	Weighting of *Fn* by a constant.
FnCustomDist(dist)	Custom distance on *Fn*, removes norm and inner.
FnCustomInnerProduct(inner[, dist_using_inner])	Custom inner product on *Fn*.
FnCustomNorm(norm)	Custom norm on *Fn*, removes inner.
FnMatrixWeighting(matrix[, exponent, ...])	Matrix weighting for *Fn*.
FnNoWeighting([exponent, dist_using_inner])	Weighting of *Fn* with constant 1.
FnVector(space, data)	Representation of an *Fn* element.
FnVectorWeighting(vector[, exponent, ...])	Vector weighting for *Fn*.
MatVecOperator(matrix[, dom, ran])	Matrix multiply operator $\mathbb{F}^n -> \mathbb{F}^m$.
Ntuples(size, dtype)	The set of n-tuples of arbitrary type.
NtuplesVector(space, data)	Representation of an *Ntuples* element.

Fn

class odl.space.ntuples.**Fn** (*size*, *dtype*, ***kwargs*)

Bases: *odl.space.base_ntuples.FnBase*, *odl.space.ntuples.Ntuples*

The vector space $\mathbb{F}^n$ with vector multiplication.

This space implements n-tuples of elements from a field $\mathbb{F}$, which can be the real or the complex numbers.

Its elements are represented as instances of the *FnVector* class.

Attributes

dtype	The data type of each entry.
element_type	*FnVector*
exponent	Exponent of the norm and distance.
field	The field of this vector space.
is_cn	If the space represents the set C^n.
is_rn	If the space represents the set R^n.
real_dtype	The corresponding real data type of this space.
size	The number of entries per tuple.

Fn.dtype

Fn.**dtype**

The data type of each entry.

Fn.element_type

Fn.**element_type**

FnVector

Fn.exponent

Fn.**exponent**

Exponent of the norm and distance.

Fn.field

Fn.**field**

The field of this vector space.

The field is the set of scalars of the space, that is numbers that the vectors in the space can be multiplied with.

Returnsfield : *Field*

The underlying field.

Fn.is_cn

Fn.**is_cn**

If the space represents the set C^n.

Tuples of complex numbers.

Fn.is_rn

Fn.**is_rn**

If the space represents the set R^n.

Tuples of real numbers.

Fn.real_dtype

`Fn.`**`real_dtype`**

The corresponding real data type of this space.

Fn.size

`Fn.`**`size`**

The number of entries per tuple.

Methods

__eq__(other)	Return `self == other`.
_dist(x1, x2)	Calculate the distance between two vectors.
_divide(x1, x2, out)	The entry-wise division of two vectors, assigned to out.
_inner(x1, x2)	Raw inner product of two vectors.
_lincomb(a, x1, b, x2, out)	Linear combination of `x1` and `x2`.
_multiply(x1, x2, out)	The entry-wise product of two vectors, assigned to out.
_norm(x)	Calculate the norm of a vector.
element([inp, data_ptr])	Create a new element.
one()	Create a vector of ones.
zero()	Create a vector of zeros.

Fn.__eq__

`Fn.`**`__eq__`**`(`*other*`)`

Return `self == other`.

Returnsequals : `bool`

`True` if other is an instance of this space's type with the same *NtuplesBase.size* and *NtuplesBase.dtype*, and identical distance function, otherwise `False`.

Examples

```
>>> from numpy.linalg import norm
>>> def dist(x, y, ord):
...     return norm(x - y, ord)
```

```
>>> from functools import partial
>>> dist2 = partial(dist, ord=2)
>>> c3 = Cn(3, dist=dist2)
>>> c3_same = Cn(3, dist=dist2)
>>> c3  == c3_same
True
```

Different `dist` functions result in different spaces - the same applies for `norm` and `inner`:

```
>>> dist1 = partial(dist, ord=1)
>>> c3_1 = Cn(3, dist=dist1)
>>> c3_2 = Cn(3, dist=dist2)
>>> c3_1 == c3_2
False
```

Be careful with Lambdas - they result in non-identical function objects:

```
>>> c3_lambda1 = Cn(3, dist=lambda x, y: norm(x-y, ord=1))
>>> c3_lambda2 = Cn(3, dist=lambda x, y: norm(x-y, ord=1))
>>> c3_lambda1 == c3_lambda2
False
```

An *Fn* space with the same data type is considered equal:

```
>>> c3 = Cn(3)
>>> f3_cdouble = Fn(3, dtype='complex128')
>>> c3 == f3_cdouble
True
```

Fn._dist

`Fn.`**`_dist`**(*x1*, *x2*)

Calculate the distance between two vectors.

Parametersx1, x2 : *FnVector*

Vectors whose mutual distance is calculated

Returnsdist : `float`

Distance between the vectors

Examples

```
>>> from numpy.linalg import norm
>>> c2_2 = Cn(2, dist=lambda x, y: norm(x - y, ord=2))
>>> x = c2_2.element([3+1j, 4])
>>> y = c2_2.element([1j, 4-4j])
>>> c2_2.dist(x, y)
5.0
```

```
>>> c2_2 = Cn(2, dist=lambda x, y: norm(x - y, ord=1))
>>> x = c2_2.element([3+1j, 4])
>>> y = c2_2.element([1j, 4-4j])
>>> c2_2.dist(x, y)
7.0
```

Fn._divide

`Fn.`**`_divide`**(*x1*, *x2*, *out*)

The entry-wise division of two vectors, assigned to out.

Parametersx1 : *FnVector*

Dividend

x1 : *FnVector*

Divisior

out : *FnVector*

The result vector, quotient

Returns`None`

Examples

```
>>> r3 = Rn(3)
>>> x = r3.element([3, 5, 6])
>>> y = r3.element([1, 2, 2])
>>> out = r3.element()
>>> r3.divide(x, y, out)  # out is returned
Rn(3).element([3.0, 2.5, 3.0])
>>> out
Rn(3).element([3.0, 2.5, 3.0])
```

Fn._inner

`Fn.__inner__(x1, x2)`

Raw inner product of two vectors.

Parametersx1, x2 : *FnVector*

The vectors whose inner product is calculated

Returnsinner : *field* element

Inner product of the vectors

Examples

```
>>> import numpy as np
>>> c3 = Cn(2, inner=lambda x, y: np.vdot(y, x))
>>> x = c3.element([5+1j, -2j])
>>> y = c3.element([1, 1+1j])
>>> c3.inner(x, y) == (5+1j)*1 + (-2j)*(1-1j)
True
```

Define a space with custom inner product:

```
>>> weights = np.array([1., 2.])
>>> def weighted_inner(x, y):
...     return np.vdot(weights * y.data, x.data)
```

```
>>> c3w = Cn(2, inner=weighted_inner)
>>> x = c3w.element(x)  # elements must be cast (no copy)
>>> y = c3w.element(y)
>>> c3w.inner(x, y) == 1*(5+1j)*1 + 2*(-2j)*(1-1j)
True
```

Fn._lincomb

`Fn.__lincomb__(a, x1, b, x2, out)`

Linear combination of `x1` and `x2`.

Calculate `out = a*x1 + b*x2` using optimized BLAS routines if possible.

Parametersa, b : *FnBase.field*

Scalar to multiply x and y with.

x1, x2 : *FnVector*

The summands

out : *FnVector*

The vector to which the result is written

ReturnsNone

Examples

```
>>> c3 = Cn(3)
>>> x = c3.element([1+1j, 2-1j, 3])
>>> y = c3.element([4+0j, 5, 6+0.5j])
>>> out = c3.element()
>>> c3.lincomb(2j, x, 3-1j, y, out)  # out is returned
Cn(3).element([(10-2j), (17-1j), (18.5+1.5j)])
>>> out
Cn(3).element([(10-2j), (17-1j), (18.5+1.5j)])
```

Fn._multiply

Fn.**_multiply**(*x1*, *x2*, *out*)

The entry-wise product of two vectors, assigned to out.

Parametersx1, x2 : *FnVector*

Factors in the product

out : *FnVector*

The result vector

ReturnsNone

Examples

```
>>> c3 = Cn(3)
>>> x = c3.element([5+1j, 3, 2-2j])
>>> y = c3.element([1, 2+1j, 3-1j])
>>> out = c3.element()
>>> c3.multiply(x, y, out)  # out is returned
Cn(3).element([(5+1j), (6+3j), (4-8j)])
>>> out
Cn(3).element([(5+1j), (6+3j), (4-8j)])
```

Fn._norm

Fn.**_norm**(*x*)

Calculate the norm of a vector.

Parametersx : *FnVector*

The vector whose norm is calculated

Returnsnorm : float

Norm of the vector

Examples

```
>>> import numpy as np
>>> c2_2 = Cn(2, norm=np.linalg.norm)  # 2-norm
>>> x = c2_2.element([3+1j, 1-5j])
>>> c2_2.norm(x)
6.0
```

```
>>> from functools import partial
>>> c2_1 = Cn(2, norm=partial(np.linalg.norm, ord=1))
>>> x = c2_1.element([3-4j, 12+5j])
>>> c2_1.norm(x)
18.0
```

Fn.element

`Fn.element(inp=None, data_ptr=None)`

Create a new element.

Parametersinp : array-like, optional

Input to initialize the new element.

If `inp` is `None`, an empty element is created with no guarantee of its state (memory allocation only).

If `inp` is a `numpy.ndarray` of shape `(size,)` and the same data type as this space, the array is wrapped, not copied. Other array-like objects are copied.

Returnselement : *NtuplesVector*

The new element created (from `inp`).

Notes

This method preserves "array views" of correct size and type, see the examples below.

Examples

```
>>> strings3 = Ntuples(3, dtype='U1')  # 1-char strings
>>> x = strings3.element(['w', 'b', 'w'])
>>> print(x)
[w, b, w]
>>> x.space
Ntuples(3, '<U1')
```

Construction from data pointer:

```
>>> int3 = Ntuples(3, dtype='int')
>>> x = int3.element([1, 2, 3])
>>> y = int3.element(data_ptr=x.data_ptr)
>>> print(y)
[1, 2, 3]
>>> y[0] = 5
>>> print(x)
[5, 2, 3]
```

Fn.one

`Fn.one()`

Create a vector of ones.

Examples

```
>>> c3 = Cn(3)
>>> x = c3.one()
>>> x
Cn(3).element([(1+0j), (1+0j), (1+0j)])
```

Fn.zero

`Fn.zero()`

Create a vector of zeros.

Examples

```
>>> c3 = Cn(3)
>>> x = c3.zero()
>>> x
Cn(3).element([0j, 0j, 0j])
```

`__init__`(*size*, *dtype*, ***kwargs*)

Initialize a new instance.

Parameterssize : positive `int`

The number of dimensions of the space

dtype : `object`

The data type of the storage array. Can be provided in any way the `numpy.dtype` function understands, most notably as built-in type, as `numpy.dtype` or as `string`.

Only scalar data types are allowed.

kwargs : {'weight', 'exponent', 'dist', 'norm', 'inner', 'dist_using_inner'}

'weight'[array-like, float or `None`] Use weighted inner product, norm, and dist.

`None` (default): No weighting, use standard functions

`float`: Weighting by a constant

array-like: Weighting by a matrix (2-dim. array) or a vector (1-dim. array, corresponds to a diagonal matrix). A matrix can also be given as a sparse matrix (`scipy.sparse.spmatrix`).

This option cannot be combined with `dist`, `norm` or `inner`.

'exponent'[positive `float`] Exponent of the norm. For values other than 2.0, no inner product is defined. If `weight` is a sparse matrix, only 1.0, 2.0 and `inf` are allowed.

This option is ignored if `dist`, `norm` or `inner` is given.

Default: 2.0

'dist'[`callable`, optional] The distance function defining a metric on $\mathbb{F}^n$. It must accept two *`FnVector`* arguments and fulfill the following mathematical conditions for any three vectors x, y, z:

- $d(x, y) = d(y, x)$
- $d(x, y) \geq 0$
- $d(x, y) = 0 \Leftrightarrow x = y$
- $d(x, y) \geq d(x, z) + d(z, y)$

By default, `dist(x, y)` is calculated as `norm(x - y)`. This creates an intermediate array `x-y`, which can be avoided by choosing `dist_using_inner=True`.

This option cannot be combined with `weight`, `norm` or `inner`.

'norm'[`callable`, optional] The norm implementation. It must accept an *FnVector* argument, return a `float` and satisfy the following conditions for all vectors x, y and scalars s:

- $\|x\| \geq 0$
- $\|x\| = 0 \Leftrightarrow x = 0$
- $\|sx\| = |s|\|x\|$
- $\|x + y\| \leq \|x\| + \|y\|$.

By default, `norm(x)` is calculated as `inner(x, x)`.

This option cannot be combined with `weight`, `dist` or `inner`.

'inner'[`callable`, optional] The inner product implementation. It must accept two *FnVector* arguments, return a element from the field of the space (real or complex number) and satisfy the following conditions for all vectors x, y, z and scalars s:

- $\langle x, y\rangle = \overline{\langle y, x\rangle}$
- $\langle sx, y\rangle = s\langle x, y\rangle$
- $\langle x + z, y\rangle = \langle x, y\rangle + \langle z, y\rangle$
- $\langle x, x\rangle = 0 \Leftrightarrow x = 0$

This option cannot be combined with `weight`, `dist` or `norm`.

dist_using_inner[`bool`, optional] Calculate `dist` using the formula

$\|x - y\|^2 = \|x\|^2 + \|y\|^2 - 2\Re\langle x, y\rangle$.

This avoids the creation of new arrays and is thus faster for large arrays. On the downside, it will not evaluate to exactly zero for equal (but not identical) x and y.

This option can only be used if `exponent` is 2.0.

Default: `False`.

FnConstWeighting

class `odl.space.ntuples.`**`FnConstWeighting`**(*constant*, *exponent=2.0*, *dist_using_inner=False*)

Bases: *odl.space.base_ntuples.FnWeightingBase*

Weighting of *Fn* by a constant.

For exponent 2.0, a new weighted inner product with constant c is defined as

$\langle a, b\rangle_c := c\, b^H a$

with b^H standing for transposed complex conjugate.

For other exponents, only norm and dist are defined. In the case of exponent `inf`, the weighted norm is defined as

$\|a\|_{c,\infty} := c\|a\|_\infty$,

otherwise it is

$\|a\|_{c,p} := c^{1/p}\|a\|_p$.

Not that this definition does **not** fulfill the limit property in p, i.e.

$\lim_{p\to\infty}\|a\|_{c,p} = \|a\|_\infty \neq \|a\|_{c,\infty}$

unless $c = 1$.

The constant c must be positive.

Attributes

const	Weighting constant of this inner product.
exponent	Exponent of this weighting.
impl	Implementation backend of this weighting.

FnConstWeighting.const
`FnConstWeighting.`**`const`**
Weighting constant of this inner product.

FnConstWeighting.exponent
`FnConstWeighting.`**`exponent`**
Exponent of this weighting.

FnConstWeighting.impl
`FnConstWeighting.`**`impl`**
Implementation backend of this weighting.

Methods

__eq__(other)	Return `self == other`.
dist(x1, x2)	Calculate the constant-weighted distance between two vectors.
equiv(other)	Test if other is an equivalent weighting.
inner(x1, x2)	Calculate the constant-weighted inner product of two vectors.
norm(x)	Calculate the constant-weighted norm of a vector.

FnConstWeighting.__eq__
`FnConstWeighting.`**`__eq__`**`(`*other*`)`
Return `self == other`.

Returnsequal : `bool`

`True` if other is an *FnConstWeighting* instance with the same constant, `False` otherwise.

FnConstWeighting.dist
`FnConstWeighting.`**`dist`**(*x1*, *x2*)
Calculate the constant-weighted distance between two vectors.

Parametersx1, x2 : *FnVector*

Vectors whose mutual distance is calculated

Returnsdist : `float`

The distance between the vectors

FnConstWeighting.equiv
`FnConstWeighting.`**`equiv`**(*other*)
Test if other is an equivalent weighting.

Returnsequivalent : `bool`

`True` if other is an *FnWeightingBase* instance with the same *FnWeightingBase.impl*, which yields the same result as this inner product for any input, `False` otherwise. This is the same as equality if `other` is an *FnConstWeighting* instance, otherwise the *equiv* method of `other` is called.

FnConstWeighting.inner
`FnConstWeighting.`**`inner`**(*x1*, *x2*)
Calculate the constant-weighted inner product of two vectors.

Parametersx1, x2 : *FnVector*

Vectors whose inner product is calculated

Returnsinner : `float` or `complex`

The inner product of the two provided vectors

FnConstWeighting.norm
`FnConstWeighting.`**`norm`**(*x*)
Calculate the constant-weighted norm of a vector.

Parametersx1 : *FnVector*

Vector whose norm is calculated

Returnsnorm : `float`

The norm of the vector
`__init__`(*constant*, *exponent=2.0*, *dist_using_inner=False*)
Initialize a new instance.

Parametersconstant : positive `float`

Weighting constant of the inner product.

exponent : positive `float`

Exponent of the norm. For values other than 2.0, the inner product is not defined.

dist_using_inner : `bool`, optional

Calculate `dist` using the formula

$\|x - y\|^2 = \|x\|^2 + \|y\|^2 - 2\Re\langle x, y\rangle$.

This avoids the creation of new arrays and is thus faster for large arrays. On the downside, it will not evaluate to exactly zero for equal (but not identical) x and y.

Can only be used if `exponent` is 2.0.

FnCustomDist

class `odl.space.ntuples.`**`FnCustomDist`**(*dist*)
Bases: *odl.space.base_ntuples.FnWeightingBase*

Custom distance on *Fn*, removes `norm` and `inner`.

Attributes

dist	Custom distance of this instance..
exponent	Exponent of this weighting.
impl	Implementation backend of this weighting.

FnCustomDist.dist
`FnCustomDist.`**`dist`**
Custom distance of this instance..

FnCustomDist.exponent
`FnCustomDist.`**`exponent`**
Exponent of this weighting.

FnCustomDist.impl
`FnCustomDist.`**`impl`**
Implementation backend of this weighting.

Methods

__eq__(other)	Return `self == other`.
inner(x1, x2)	Inner product is not defined for custom distance.
norm(x)	Norm is not defined for custom distance.

FnCustomDist.__eq__
`FnCustomDist.`**`__eq__`**(*other*)
Return `self == other`.

Returnsequal : `bool`

`True` if other is an *FnCustomDist* instance with the same norm, `False` otherwise.

FnCustomDist.inner
`FnCustomDist.`**`inner`**(*x1*, *x2*)
Inner product is not defined for custom distance.

FnCustomDist.norm
FnCustomDist.**norm**(*x*)
Norm is not defined for custom distance.
__init__(*dist*)
Initialize a new instance.

Parametersdist : callable

The distance function defining a metric on $\mathbb{F}^n$. It must accept two *FnVector* arguments and fulfill the following mathematical conditions for any three vectors x, y, z:

- $d(x, y) = d(y, x)$
- $d(x, y) \geq 0$
- $d(x, y) = 0 \Leftrightarrow x = y$
- $d(x, y) \geq d(x, z) + d(z, y)$

FnCustomInnerProduct

class odl.space.ntuples.**FnCustomInnerProduct**(*inner*, *dist_using_inner=False*)
Bases: *odl.space.base_ntuples.FnWeightingBase*

Custom inner product on *Fn*.

Attributes

exponent	Exponent of this weighting.
impl	Implementation backend of this weighting.
inner	Custom inner product of this instance..

FnCustomInnerProduct.exponent
FnCustomInnerProduct.**exponent**
Exponent of this weighting.

FnCustomInnerProduct.impl
FnCustomInnerProduct.**impl**
Implementation backend of this weighting.

FnCustomInnerProduct.inner
FnCustomInnerProduct.**inner**
Custom inner product of this instance..

Methods

__eq__(other)	Return `self == other`.

FnCustomInnerProduct.__eq__
FnCustomInnerProduct.**__eq__**(*other*)
Return `self == other`.

Returnsequal : `bool`

`True` if other is an *FnCustomInnerProduct* instance with the same inner product, `False` otherwise.

__init__(*inner*, *dist_using_inner=False*)

Initialize a new instance.

Parametersinner : `callable`

The inner product implementation. It must accept two *FnVector* arguments, return a element from the field of the space (real or complex number) and satisfy the following conditions for all vectors x, y, z and scalars s:

- $\langle x, y\rangle = \overline{\langle y, x\rangle}$
- $\langle sx, y\rangle = s\langle x, y\rangle$
- $\langle x + z, y\rangle = \langle x, y\rangle + \langle z, y\rangle$
- $\langle x, x\rangle = 0 \Leftrightarrow x = 0$

dist_using_inner : `bool`, optional

Calculate `dist` using the formula

$\|x - y\|^2 = \|x\|^2 + \|y\|^2 - 2\Re\langle x, y\rangle$.

This avoids the creation of new arrays and is thus faster for large arrays. On the downside, it will not evaluate to exactly zero for equal (but not identical) x and y.

FnCustomNorm

class odl.space.ntuples.**FnCustomNorm**(*norm*)

Bases: *odl.space.base_ntuples.FnWeightingBase*

Custom norm on *Fn*, removes `inner`.

Attributes

exponent	Exponent of this weighting.
impl	Implementation backend of this weighting.
norm	Custom norm of this instance..

FnCustomNorm.exponent

FnCustomNorm.**exponent**

Exponent of this weighting.

FnCustomNorm.impl

FnCustomNorm.**impl**

Implementation backend of this weighting.

FnCustomNorm.norm

FnCustomNorm.**norm**

Custom norm of this instance..

Methods

__eq__(other)	Return `self == other`.
inner(x1, x2)	Inner product is not defined for custom distance.

FnCustomNorm.__eq__
`FnCustomNorm.__eq__`(*other*)
Return `self == other`.

Returnsequal : `bool`

`True` if other is an *FnCustomNorm* instance with the same norm, `False` otherwise.

FnCustomNorm.inner
`FnCustomNorm.inner`(*x1*, *x2*)
Inner product is not defined for custom distance.
`__init__`(*norm*)
Initialize a new instance.

Parametersnorm : `callable`

The norm implementation. It must accept an *FnVector* argument, return a `float` and satisfy the following conditions for all vectors x, y and scalars s:

- $\|x\| \geq 0$
- $\|x\| = 0 \Leftrightarrow x = 0$
- $\|sx\| = |s|\|x\|$
- $\|x + y\| \leq \|x\| + \|y\|$.

FnMatrixWeighting

class `odl.space.ntuples.`**`FnMatrixWeighting`**(*matrix*, *exponent=2.0*, *dist_using_inner=False*, ***kwargs*)

Bases: *odl.space.base_ntuples.FnWeightingBase*

Matrix weighting for *Fn*.

For exponent 2.0, a new weighted inner product with matrix W is defined as

$\langle a, b\rangle_W := b^H W a$

with b^H standing for transposed complex conjugate.

For other exponents, only norm and dist are defined. In the case of exponent `inf`, the weighted norm is

$\|a\|_{W,\infty} := \|Wa\|_\infty$,

otherwise it is

$\|a\|_{W,p} := \|W^{1/p} a\|_p$.

Not that this definition does **not** fulfill the limit property in p, i.e.

$\lim_{p\to\infty} \|a\|_{W,p} = \|a\|_\infty \neq \|a\|_{W,\infty}$

unless W is the identity matrix.

The matrix must be Hermitian and posivive definite, otherwise it does not define an inner product or norm, respectively. This is not checked during initialization.

Attributes

exponent	Exponent of this weighting.
impl	Implementation backend of this weighting.
matrix	Weighting matrix of this inner product.
matrix_issparse	Whether the representing matrix is sparse or not.

FnMatrixWeighting.exponent
`FnMatrixWeighting.`**`exponent`**
Exponent of this weighting.

FnMatrixWeighting.impl
`FnMatrixWeighting.`**`impl`**
Implementation backend of this weighting.

FnMatrixWeighting.matrix
`FnMatrixWeighting.`**`matrix`**
Weighting matrix of this inner product.

FnMatrixWeighting.matrix_issparse
`FnMatrixWeighting.`**`matrix_issparse`**
Whether the representing matrix is sparse or not.

Methods

__eq__(other)	Return `self == other`.
equiv(other)	Test if other is an equivalent weighting.
inner(x1, x2)	Calculate the matrix-weighted inner product of two vectors.
matrix_isvalid()	Test if the matrix is positive definite Hermitian.
norm(x)	Calculate the matrix-weighted norm of a vector.

FnMatrixWeighting.__eq__
`FnMatrixWeighting.`**`__eq__`**(*other*)
Return `self == other`.

Returnsequals : `bool`

`True` if other is an *FnMatrixWeighting* instance with **identical** matrix, `False` otherwise.

See also:

equivtest for equivalent inner products

FnMatrixWeighting.equiv

FnMatrixWeighting.**equiv**(*other*)
Test if other is an equivalent weighting.

Returnsequivalent : bool

True if other is an *FnWeightingBase* instance with the same *FnWeightingBase.impl*, which yields the same result as this inner product for any input, False otherwise. This is checked by entry-wise comparison of this inner product's matrix with the matrix or constant of other.

FnMatrixWeighting.inner
FnMatrixWeighting.**inner**(*x1*, *x2*)
Calculate the matrix-weighted inner product of two vectors.

Parametersx1, x2 : *FnVector*

Vectors whose inner product is calculated

Returnsinner : float or complex

The inner product of the vectors

FnMatrixWeighting.matrix_isvalid
FnMatrixWeighting.**matrix_isvalid**()
Test if the matrix is positive definite Hermitian.

This test tries to calculate a Cholesky decomposition and can be very time-consuming for large matrices. Sparse matrices are not supported.

FnMatrixWeighting.norm
FnMatrixWeighting.**norm**(*x*)
Calculate the matrix-weighted norm of a vector.

Parametersx : *FnVector*

Vector whose norm is calculated

Returnsnorm : float

The norm of the vector

__init__(*matrix*, *exponent=2.0*, *dist_using_inner=False*, ***kwargs*)
Initialize a new instance.

Parametersmatrix : scipy.sparse.spmatrix or array-like, 2-dim.

Square weighting matrix of the inner product

exponent : positive float

Exponent of the norm. For values other than 2.0, the inner product is not defined. If matrix is a sparse matrix, only 1.0, 2.0 and inf are allowed.

dist_using_inner : bool, optional

Calculate dist using the formula

$\|x - y\|^2 = \|x\|^2 + \|y\|^2 - 2\Re\langle x, y\rangle$.

This avoids the creation of new arrays and is thus faster for large arrays. On the downside, it will not evaluate to exactly zero for equal (but not identical) x and y.

Can only be used if exponent is 2.0.

kwargs : {'precomp_mat_pow', 'cache_mat_pow'}

'precomp_mat_pow'[bool] If `True`, precompute the matrix power $W^{1/p}$ during initialization. This has no effect if `exponent` is 1.0, 2.0 or `inf`.

Default: `False`

'cache_mat_pow'[bool] If `True`, cache the matrix power $W^{1/p}$ during the first call to `norm` or `dist`. This has no effect if `exponent` is 1.0, 2.0 or `inf`.

Default: `False`

FnNoWeighting

class odl.space.ntuples.**FnNoWeighting**(*exponent=2.0*, *dist_using_inner=False*)

Bases: *odl.space.ntuples.FnConstWeighting*

Weighting of *Fn* with constant 1.

For exponent 2.0, the unweighted inner product is defined as

$\langle a, b \rangle := b^H a$

with b^H standing for transposed complex conjugate.

For other exponents, only norm and dist are defined.

Attributes

const	Weighting constant of this inner product.
exponent	Exponent of this weighting.
impl	Implementation backend of this weighting.

FnNoWeighting.const

FnNoWeighting.**const**

Weighting constant of this inner product.

FnNoWeighting.exponent

FnNoWeighting.**exponent**

Exponent of this weighting.

FnNoWeighting.impl

FnNoWeighting.**impl**

Implementation backend of this weighting.

Methods

__eq__(other)	Return `self == other`.
dist(x1, x2)	Calculate the constant-weighted distance between two vectors.
equiv(other)	Test if other is an equivalent weighting.
inner(x1, x2)	Calculate the constant-weighted inner product of two vectors.
norm(x)	Calculate the constant-weighted norm of a vector.

FnNoWeighting.__eq__
FnNoWeighting.**__eq__**(*other*)
Return `self == other`.

Returnsequal : `bool`

`True` if other is an *FnConstWeighting* instance with the same constant, `False` otherwise.

FnNoWeighting.dist
FnNoWeighting.**dist**(*x1*, *x2*)
Calculate the constant-weighted distance between two vectors.

Parametersx1, x2 : *FnVector*

Vectors whose mutual distance is calculated

Returnsdist : `float`

The distance between the vectors

FnNoWeighting.equiv
FnNoWeighting.**equiv**(*other*)
Test if other is an equivalent weighting.

Returnsequivalent : `bool`

`True` if other is an *FnWeightingBase* instance with the same *FnWeightingBase.impl*, which yields the same result as this inner product for any input, `False` otherwise. This is the same as equality if `other` is an *FnConstWeighting* instance, otherwise the *equiv* method of `other` is called.

FnNoWeighting.inner
FnNoWeighting.**inner**(*x1*, *x2*)
Calculate the constant-weighted inner product of two vectors.

Parametersx1, x2 : *FnVector*

Vectors whose inner product is calculated

Returnsinner : `float` or `complex`

The inner product of the two provided vectors

FnNoWeighting.norm
FnNoWeighting.**norm**(*x*)
Calculate the constant-weighted norm of a vector.

Parametersx1 : *FnVector*

Vector whose norm is calculated

Returnsnorm : `float`

The norm of the vector
__init__(*exponent=2.0*, *dist_using_inner=False*)
Initialize a new instance.

Parametersexponent : positive `float`

Exponent of the norm. For values other than 2.0, the inner product is not defined.

dist_using_inner : `bool`, optional

Calculate `dist` using the formula

$\|x-y\|^2 = \|x\|^2 + \|y\|^2 - 2\Re\langle x, y\rangle$.

This avoids the creation of new arrays and is thus faster for large arrays. On the downside, it will not evaluate to exactly zero for equal (but not identical) x and y.

Can only be used if `exponent` is 2.0.

FnVector

class `odl.space.ntuples.`**`FnVector`**(*space*, *data*)

Bases: *odl.space.base_ntuples.FnBaseVector*, *odl.space.ntuples.NtuplesVector*

Representation of an *Fn* element.

Attributes

T	The transpose of a vector, the functional given by (.
data	The raw `numpy.ndarray` representing the data.
data_ptr	A raw pointer to the data container.
dtype	Length of this vector, equal to space size.
imag	The imaginary part of this vector.
itemsize	The size in bytes on one element of this type.
nbytes	The number of bytes this vector uses in memory.
ndim	Number of dimensions, always 1.
real	The real part of this vector.
shape	Shape of this vector, equals `(size,)`.
size	Length of this vector, equal to space size.
space	Space to which this vector.
ufunc	*NtuplesUFuncs*, access to numpy style ufuncs.

FnVector.T

`FnVector.`**`T`**

The transpose of a vector, the functional given by (. , self)

Returnstranspose : *InnerProductOperator*

Notes

This function is only defined in inner product spaces.

In a complex space, this takes the conjugate transpose of the vector.

Examples

```
>>> from odl import Rn
>>> import numpy as np
```

```
>>> rn = Rn(3)
>>> x = rn.element([1, 2, 3])
>>> y = rn.element([2, 1, 3])
>>> x.T(y)
13.0
```

FnVector.data

`FnVector.data`

The raw `numpy.ndarray` representing the data.

FnVector.data_ptr

`FnVector.data_ptr`

A raw pointer to the data container.

Examples

```
>>> import ctypes
>>> vec = Ntuples(3, 'int32').element([1, 2, 3])
>>> arr_type = ctypes.c_int32 * 3
>>> buffer = arr_type.from_address(vec.data_ptr)
>>> arr = np.frombuffer(buffer, dtype='int32')
>>> print(arr)
[1 2 3]
```

In-place modification via pointer:

```
>>> arr[0] = 5
>>> print(vec)
[5, 2, 3]
```

FnVector.dtype

`FnVector.dtype`

Length of this vector, equal to space size.

FnVector.imag

`FnVector.imag`

The imaginary part of this vector.

Returnsimag : *FnVector*

The imaginary part this vector as a vector in *Rn*

Examples

```
>>> c3 = Cn(3)
>>> x = c3.element([5+1j, 3, 2-2j])
>>> x.imag
Rn(3).element([1.0, 0.0, -2.0])
```

The *Rn* vector is really a view, so changes affect the original array:

```
>>> x.imag *= 2
>>> x
Cn(3).element([(5+2j), (3+0j), (2-4j)])
```

FnVector.itemsize
`FnVector.`**`itemsize`**
The size in bytes on one element of this type.

FnVector.nbytes
`FnVector.`**`nbytes`**
The number of bytes this vector uses in memory.

FnVector.ndim
`FnVector.`**`ndim`**
Number of dimensions, always 1.

FnVector.real
`FnVector.`**`real`**
The real part of this vector.

Returnsreal : *FnVector* view with dtype

The real part this vector as a vector in *Rn*

Examples

```
>>> c3 = Cn(3)
>>> x = c3.element([5+1j, 3, 2-2j])
>>> x.real
Rn(3).element([5.0, 3.0, 2.0])
```

The *Rn* vector is really a view, so changes affect the original array:

```
>>> x.real *= 2
>>> x
Cn(3).element([(10+1j), (6+0j), (4-2j)])
```

FnVector.shape
`FnVector.`**`shape`**
Shape of this vector, equals `(size,)`.

FnVector.size
`FnVector.`**`size`**
Length of this vector, equal to space size.

FnVector.space
`FnVector.`**`space`**
Space to which this vector.

FnVector.ufunc

FnVector.**ufunc**

NtuplesUFuncs, access to numpy style ufuncs.

Notes

These are optimized for use with ntuples and incur no overhead.

Examples

```
>>> r2 = Rn(2)
>>> x = r2.element([1, -2])
>>> x.ufunc.absolute()
Rn(2).element([1.0, 2.0])
```

These functions can also be used with broadcasting

```
>>> x.ufunc.add(3)
Rn(2).element([4.0, 1.0])
```

and non-space elements

```
>>> x.ufunc.subtract([3, 3])
Rn(2).element([-2.0, -5.0])
```

There is also support for various reductions (sum, prod, min, max)

```
>>> x.ufunc.sum()
-1.0
```

They also support an out parameter

```
>>> y = r2.element([3, 4])
>>> out = r2.element()
>>> result = x.ufunc.add(y, out=out)
>>> result
Rn(2).element([4.0, 2.0])
>>> result is out
True
```

Methods

__getitem__(indices)	Access values of this vector.
__setitem__(indices, values)	Set values of this vector.
asarray([start, stop, step, out])	Extract the data of this array as a numpy array.
conj([out])	The complex conjugate of this vector.

FnVector.__getitem__

FnVector.**__getitem__**(*indices*)

Access values of this vector.

Parametersindices : int or slice

The position(s) that should be accessed

Returnsvalues : scalar or *NtuplesVector*

The value(s) at the index (indices)

Examples

```
>>> str_3 = Ntuples(3, dtype='U6')  # 6-char unicode
>>> x = str_3.element(['a', 'Hello!', '0'])
>>> print(x[0])
a
>>> print(x[1:3])
[Hello!, 0]
>>> x[1:3].space
Ntuples(2, '<U6')
```

FnVector.__setitem__

`FnVector.__setitem__`(*indices*, *values*)

Set values of this vector.

Parametersindices : `int` or `slice`

The position(s) that should be set

values : {scalar, array-like, *NtuplesVector*}

The value(s) that are to be assigned.

If `indices` is an integer, `value` must be scalar.

If `indices` is a slice, `value` must be broadcastable to the size of the slice (same size, shape `(1,)` or single value).

Returns`None`

Examples

```
>>> int_3 = Ntuples(3, 'int')
>>> x = int_3.element([1, 2, 3])
>>> x[0] = 5
>>> x
Ntuples(3, 'int').element([5, 2, 3])
```

Assignment from array-like structures or another vector:

```
>>> y = Ntuples(2, 'short').element([-1, 2])
>>> x[:2] = y
>>> x
Ntuples(3, 'int').element([-1, 2, 3])
>>> x[1:3] = [7, 8]
>>> x
Ntuples(3, 'int').element([-1, 7, 8])
>>> x[:] = np.array([0, 0, 0])
>>> x
Ntuples(3, 'int').element([0, 0, 0])
```

Broadcasting is also supported:

```
>>> x[1:3] = -2.
>>> x
Ntuples(3, 'int').element([0, -2, -2])
```

Array views are preserved:

```
>>> y = x[::2]  # view into x
>>> y[:] = -9
>>> print(y)
[-9, -9]
>>> print(x)
[-9, -2, -9]
```

Be aware of unsafe casts and over-/underflows, there will be warnings at maximum.

```
>>> x = Ntuples(2, 'int8').element([0, 0])
>>> maxval = 255  # maximum signed 8-bit unsigned int
>>> x[0] = maxval + 1
>>> x
Ntuples(2, 'int8').element([0, 0])
```

FnVector.asarray

`FnVector.`**`asarray`**(*start=None*, *stop=None*, *step=None*, *out=None*)

Extract the data of this array as a numpy array.

Parametersstart : `int`, optional

Start position. None means the first element.

start : `int`, optional

One element past the last element to be extracted. None means the last element.

start : `int`, optional

Step length. None means 1.

out : `numpy.ndarray`, optional

Array in which the result should be written in-place. Has to be contiguous and of the correct dtype.

Returnsasarray : `numpy.ndarray`

Numpy array of the same type as the space.

Examples

```
>>> import ctypes
>>> vec = Ntuples(3, 'float').element([1, 2, 3])
>>> vec.asarray()
array([ 1.,  2.,  3.])
>>> vec.asarray(start=1, stop=3)
array([ 2.,  3.])
```

Using the out parameter

```
>>> out = np.empty((3,), dtype='float')
>>> result = vec.asarray(out=out)
>>> out
```

```
        array([ 1.,  2.,  3.])
        >>> result is out
        True
```

FnVector.conj

`FnVector.conj`(*out=None*)

The complex conjugate of this vector.

Parametersout : *FnVector*, optional

Vector to which the complex conjugate is written. Must be an element of this vector's space.

Returnsout : *FnVector*

The complex conjugate vector. If `out` was provided, it is returned. Otherwise, the complex conjugate is returned as a new vector.

Examples

```
        >>> x = Cn(3).element([5+1j, 3, 2-2j])
        >>> y = x.conj(); print(y)
        [(5-1j), (3-0j), (2+2j)]
```

The out parameter allows you to avoid a copy

```
        >>> z = Cn(3).element()
        >>> z_out = x.conj(out=z); print(z)
        [(5-1j), (3-0j), (2+2j)]
        >>> z_out is z
        True
```

It can also be used for inplace conj >>> x_out = x.conj(out=x); print(x) [(5-1j), (3-0j), (2+2j)] >>> x_out is x True

__init__(*space*, *data*)

Initialize a new instance.

FnVectorWeighting

class `odl.space.ntuples.`**`FnVectorWeighting`**(*vector*, *exponent=2.0*, *dist_using_inner=False*)

Bases: *odl.space.base_ntuples.FnWeightingBase*

Vector weighting for *Fn*.

For exponent 2.0, a new weighted inner product with vector w is defined as

$\langle a, b \rangle_w := b^H (w \odot a)$

with b^H standing for transposed complex conjugate and $w \odot a$ for element-wise multiplication.

For other exponents, only norm and dist are defined. In the case of exponent `inf`, the weighted norm is

$\|a\|_{w,\infty} := \|w \odot a\|_\infty$,

otherwise it is

$\|a\|_{w,p} := \|w^{1/p} \odot a\|_p$.

Not that this definition does **not** fulfill the limit property in p, i.e.

$\lim_{p \to \infty} \|a\|_{w,p} = \|a\|_\infty \neq \|a\|_{w,\infty}$

unless $w = (1, \ldots, 1)$.

The vector may only have positive entries, otherwise it does not define an inner product or norm, respectively. This is not checked during initialization.

Attributes

exponent	Exponent of this weighting.
impl	Implementation backend of this weighting.
vector	Weighting vector of this inner product.

FnVectorWeighting.exponent
`FnVectorWeighting.`**`exponent`**
Exponent of this weighting.

FnVectorWeighting.impl
`FnVectorWeighting.`**`impl`**
Implementation backend of this weighting.

FnVectorWeighting.vector
`FnVectorWeighting.`**`vector`**
Weighting vector of this inner product.

Methods

__eq__(other)	Return `self == other`.
equiv(other)	Test if other is an equivalent weighting.
inner(x1, x2)	Calculate the vector weighted inner product of two vectors.
norm(x)	Calculate the vector-weighted norm of a vector.
vector_is_valid()	Test if the vector is a valid weight, i.e.

FnVectorWeighting.__eq__
`FnVectorWeighting.`**`__eq__`**(*other*)
Return `self == other`.

Returnsequals : `bool`

`True` if other is an *FnVectorWeighting* instance with **identical** vector, `False` otherwise.

See also:

equivtest for equivalent inner products

FnVectorWeighting.equiv
`FnVectorWeighting.`**`equiv`**(*other*)
Test if other is an equivalent weighting.

Returnsequivalent : `bool`

`True` if `other` is an *`FnWeightingBase`* instance with the same *`FnWeightingBase.impl`*, which yields the same result as this inner product for any input, `False` otherwise. This is checked by entry-wise comparison of matrices/vectors/constant of this inner product and `other`.

FnVectorWeighting.inner

`FnVectorWeighting.`**`inner`**(*x1*, *x2*)

Calculate the vector weighted inner product of two vectors.

Parametersx1, x2 : *`FnVector`*

Vectors whose inner product is calculated

Returnsinner : `float` or `complex`

The inner product of the two provided vectors

FnVectorWeighting.norm

`FnVectorWeighting.`**`norm`**(*x*)

Calculate the vector-weighted norm of a vector.

Parametersx : *`FnVector`*

Vector whose norm is calculated

Returnsnorm : `float`

The norm of the provided vector

FnVectorWeighting.vector_is_valid

`FnVectorWeighting.`**`vector_is_valid`**()

Test if the vector is a valid weight, i.e. positive.

`__init__`(*vector*, *exponent=2.0*, *dist_using_inner=False*)

Initialize a new instance.

Parametersvector : array-like, one-dim.

Weighting vector of the inner product

exponent : positive `float`

Exponent of the norm. For values other than 2.0, the inner product is not defined. If `matrix` is a sparse matrix, only 1.0, 2.0 and `inf` are allowed.

dist_using_inner : `bool`, optional

Calculate `dist` using the formula

$\|x - y\|^2 = \|x\|^2 + \|y\|^2 - 2\Re\langle x, y\rangle$.

This avoids the creation of new arrays and is thus faster for large arrays. On the downside, it will not evaluate to exactly zero for equal (but not identical) x and y.

Can only be used if `exponent` is 2.0.

MatVecOperator

class odl.space.ntuples.**MatVecOperator**(*matrix*, *dom=None*, *ran=None*)

Bases: *odl.operator.operator.Operator*

Matrix multiply operator $\mathbb{F}^n -> \mathbb{F}^m$.

Attributes

adjoint	Adjoint operator represented by the adjoint matrix.
domain	The domain of this operator.
inverse	Return the operator inverse.
is_functional	`True` if the this operator's range is a *Field*.
is_linear	`True` if this operator is linear.
matrix	Matrix representing this operator.
matrix_issparse	Whether the representing matrix is sparse or not.
range	The range of this operator.

MatVecOperator.adjoint

MatVecOperator.**adjoint**

Adjoint operator represented by the adjoint matrix.

MatVecOperator.domain

MatVecOperator.**domain**

The domain of this operator.

MatVecOperator.inverse

MatVecOperator.**inverse**

Return the operator inverse.

RaisesOpNotImplementedError

Since the inverse cannot be default implemented.

MatVecOperator.is_functional

MatVecOperator.**is_functional**

`True` if the this operator's range is a *Field*.

MatVecOperator.is_linear

MatVecOperator.**is_linear**

`True` if this operator is linear.

MatVecOperator.matrix

MatVecOperator.**matrix**

Matrix representing this operator.

MatVecOperator.matrix_issparse

MatVecOperator.**matrix_issparse**

Whether the representing matrix is sparse or not.

MatVecOperator.range
`MatVecOperator.`**`range`**
The range of this operator.

Methods

`__eq__`	Return self==value.
`_apply`(x, out)	Raw apply method on input, writing to given output.
`_call`(x)	Raw call method on input, producing a new output.

MatVecOperator._apply
`MatVecOperator.`**`_apply`**(*x*, *out*)
Raw apply method on input, writing to given output.

MatVecOperator._call
`MatVecOperator.`**`_call`**(*x*)
Raw call method on input, producing a new output.
`__init__`(*matrix*, *dom=None*, *ran=None*)
Initialize a new instance.

Parametersmatrix : array-like or `scipy.sparse.spmatrix`

Matrix representing the linear operator. Its shape must be `(m, n)`, where `n` is the size of `dom` and `m` the size of `ran`. Its dtype must be castable to the range `dtype`.

dom : *`Fn`*, optional

Space on whose elements the matrix acts. If not provided, the domain is inferred from the matrix `dtype` and `shape`. If provided, its dtype must be castable to the range dtype.

ran : *`Fn`*, optional

Space to which the matrix maps. If not provided, the domain is inferred from the matrix `dtype` and `shape`.

Ntuples

class `odl.space.ntuples.`**`Ntuples`**(*size*, *dtype*)
Bases: *`odl.space.base_ntuples.NtuplesBase`*

The set of n-tuples of arbitrary type.

Attributes

`dtype`	The data type of each entry.
`element_type`	*`NtuplesVector`*
`size`	The number of entries per tuple.

Ntuples.dtype
`Ntuples.`**`dtype`**

The data type of each entry.

Ntuples.element_type
`Ntuples.element_type`
NtuplesVector

Ntuples.size
`Ntuples.size`
The number of entries per tuple.

Methods

element([inp, data_ptr])	Create a new element.

Ntuples.element
`Ntuples.element`(*inp=None*, *data_ptr=None*)
Create a new element.

Parametersinp : array-like, optional

Input to initialize the new element.

If `inp` is `None`, an empty element is created with no guarantee of its state (memory allocation only).

If `inp` is a `numpy.ndarray` of shape `(size,)` and the same data type as this space, the array is wrapped, not copied. Other array-like objects are copied.

Returnselement : *NtuplesVector*

The new element created (from `inp`).

Notes

This method preserves "array views" of correct size and type, see the examples below.

Examples

```
>>> strings3 = Ntuples(3, dtype='U1')  # 1-char strings
>>> x = strings3.element(['w', 'b', 'w'])
>>> print(x)
[w, b, w]
>>> x.space
Ntuples(3, '<U1')
```

Construction from data pointer:

```
>>> int3 = Ntuples(3, dtype='int')
>>> x = int3.element([1, 2, 3])
>>> y = int3.element(data_ptr=x.data_ptr)
>>> print(y)
[1, 2, 3]
>>> y[0] = 5
```

```
>>> print(x)
[5, 2, 3]
```

__init__(*size*, *dtype*)
Initialize a new instance.

Parameters

size : non-negative int

The number of entries per tuple

dtype : `object`

The data type for each tuple entry. Can be provided in any way the `numpy.dtype` function understands, most notably as built-in type, as one of NumPy's internal datatype objects or as string.

NtuplesVector

class odl.space.ntuples.**NtuplesVector**(*space*, *data*)
Bases: *odl.space.base_ntuples.NtuplesBaseVector*

Representation of an *Ntuples* element.

Attributes

data	The raw `numpy.ndarray` representing the data.
data_ptr	A raw pointer to the data container.
dtype	Length of this vector, equal to space size.
itemsize	The size in bytes on one element of this type.
nbytes	The number of bytes this vector uses in memory.
ndim	Number of dimensions, always 1.
shape	Shape of this vector, equals `(size,)`.
size	Length of this vector, equal to space size.
space	Space to which this vector.
ufunc	*NtuplesUFuncs*, access to numpy style ufuncs.

NtuplesVector.data
NtuplesVector.**data**
The raw `numpy.ndarray` representing the data.

NtuplesVector.data_ptr
NtuplesVector.**data_ptr**
A raw pointer to the data container.

Examples

```
>>> import ctypes
>>> vec = Ntuples(3, 'int32').element([1, 2, 3])
>>> arr_type = ctypes.c_int32 * 3
>>> buffer = arr_type.from_address(vec.data_ptr)
>>> arr = np.frombuffer(buffer, dtype='int32')
```

```
>>> print(arr)
[1 2 3]
```

In-place modification via pointer:

```
>>> arr[0] = 5
>>> print(vec)
[5, 2, 3]
```

NtuplesVector.dtype
`NtuplesVector.dtype`
Length of this vector, equal to space size.

NtuplesVector.itemsize
`NtuplesVector.itemsize`
The size in bytes on one element of this type.

NtuplesVector.nbytes
`NtuplesVector.nbytes`
The number of bytes this vector uses in memory.

NtuplesVector.ndim
`NtuplesVector.ndim`
Number of dimensions, always 1.

NtuplesVector.shape
`NtuplesVector.shape`
Shape of this vector, equals `(size,)`.

NtuplesVector.size
`NtuplesVector.size`
Length of this vector, equal to space size.

NtuplesVector.space
`NtuplesVector.space`
Space to which this vector.

NtuplesVector.ufunc
`NtuplesVector.ufunc`
`NtuplesUFuncs`, access to numpy style ufuncs.

Notes

These are optimized for use with ntuples and incur no overhead.

Examples

```
>>> r2 = Rn(2)
>>> x = r2.element([1, -2])
>>> x.ufunc.absolute()
Rn(2).element([1.0, 2.0])
```

These functions can also be used with broadcasting

```
>>> x.ufunc.add(3)
Rn(2).element([4.0, 1.0])
```

and non-space elements

```
>>> x.ufunc.subtract([3, 3])
Rn(2).element([-2.0, -5.0])
```

There is also support for various reductions (sum, prod, min, max)

```
>>> x.ufunc.sum()
-1.0
```

They also support an out parameter

```
>>> y = r2.element([3, 4])
>>> out = r2.element()
>>> result = x.ufunc.add(y, out=out)
>>> result
Rn(2).element([4.0, 2.0])
>>> result is out
True
```

Methods

__eq__(other)	Return `self == other`.
__getitem__(indices)	Access values of this vector.
__setitem__(indices, values)	Set values of this vector.
asarray([start, stop, step, out])	Extract the data of this array as a numpy array.
copy()	Create an identical (deep) copy of this vector.

NtuplesVector.__eq__

`NtuplesVector.__eq__`(*other*)

Return `self == other`.

Returnsequals : `bool`

`True` if all entries of other are equal to this vector's entries, `False` otherwise.

Notes

Space membership is not checked, hence vectors from different spaces can be equal.

Examples

```
>>> vec1 = Ntuples(3, int).element([1, 2, 3])
>>> vec2 = Ntuples(3, int).element([-1, 2, 0])
>>> vec1 == vec2
False
>>> vec2 = Ntuples(3, int).element([1, 2, 3])
>>> vec1 == vec2
True
```

Space membership matters:

```
>>> vec2 = Ntuples(3, float).element([1, 2, 3])
>>> vec1 == vec2 or vec2 == vec1
False
```

NtuplesVector.__getitem__

NtuplesVector.**__getitem__**(*indices*)

Access values of this vector.

Parametersindices : int or slice

The position(s) that should be accessed

Returnsvalues : scalar or *NtuplesVector*

The value(s) at the index (indices)

Examples

```
>>> str_3 = Ntuples(3, dtype='U6')  # 6-char unicode
>>> x = str_3.element(['a', 'Hello!', '0'])
>>> print(x[0])
a
>>> print(x[1:3])
[Hello!, 0]
>>> x[1:3].space
Ntuples(2, '<U6')
```

NtuplesVector.__setitem__

NtuplesVector.**__setitem__**(*indices*, *values*)

Set values of this vector.

Parametersindices : int or slice

The position(s) that should be set

values : {scalar, array-like, *NtuplesVector*}

The value(s) that are to be assigned.

If indices is an integer, value must be scalar.

If indices is a slice, value must be broadcastable to the size of the slice (same size, shape (1,) or single value).

ReturnsNone

Examples

```
>>> int_3 = Ntuples(3, 'int')
>>> x = int_3.element([1, 2, 3])
>>> x[0] = 5
>>> x
Ntuples(3, 'int').element([5, 2, 3])
```

Assignment from array-like structures or another vector:

```
>>> y = Ntuples(2, 'short').element([-1, 2])
>>> x[:2] = y
>>> x
Ntuples(3, 'int').element([-1, 2, 3])
>>> x[1:3] = [7, 8]
>>> x
Ntuples(3, 'int').element([-1, 7, 8])
>>> x[:] = np.array([0, 0, 0])
>>> x
Ntuples(3, 'int').element([0, 0, 0])
```

Broadcasting is also supported:

```
>>> x[1:3] = -2.
>>> x
Ntuples(3, 'int').element([0, -2, -2])
```

Array views are preserved:

```
>>> y = x[::2]  # view into x
>>> y[:] = -9
>>> print(y)
[-9, -9]
>>> print(x)
[-9, -2, -9]
```

Be aware of unsafe casts and over-/underflows, there will be warnings at maximum.

```
>>> x = Ntuples(2, 'int8').element([0, 0])
>>> maxval = 255  # maximum signed 8-bit unsigned int
>>> x[0] = maxval + 1
>>> x
Ntuples(2, 'int8').element([0, 0])
```

NtuplesVector.asarray

`NtuplesVector.`**`asarray`**(*start=None*, *stop=None*, *step=None*, *out=None*)

Extract the data of this array as a numpy array.

Parametersstart : `int`, optional

Start position. None means the first element.

start : `int`, optional

One element past the last element to be extracted. None means the last element.

start : `int`, optional

Step length. None means 1.

out : `numpy.ndarray`, optional

Array in which the result should be written in-place. Has to be contiguous and of the correct dtype.

Returnsasarray : `numpy.ndarray`

Numpy array of the same type as the space.

Examples

```
>>> import ctypes
>>> vec = Ntuples(3, 'float').element([1, 2, 3])
>>> vec.asarray()
array([ 1.,  2.,  3.])
>>> vec.asarray(start=1, stop=3)
array([ 2.,  3.])
```

Using the out parameter

```
>>> out = np.empty((3,), dtype='float')
>>> result = vec.asarray(out=out)
>>> out
array([ 1.,  2.,  3.])
>>> result is out
True
```

NtuplesVector.copy

`NtuplesVector.copy()`

Create an identical (deep) copy of this vector.

ParametersNone

Returnscopy : *NtuplesVector*

The deep copy

Examples

```
>>> vec1 = Ntuples(3, 'int').element([1, 2, 3])
>>> vec2 = vec1.copy()
>>> vec2
Ntuples(3, 'int').element([1, 2, 3])
>>> vec1 == vec2
True
>>> vec1 is vec2
False
```

__init__ (*space*, *data*)

Initialize a new instance.

Functions

Cn(size[, dtype])	The complex vector space C^n with vector multiplication.
Rn(size[, dtype])	The real vector space R^n with vector multiplication.
weighted_dist(weight[, exponent, use_inner])	Weighted distance on *Fn* spaces as free function.

Continued on next page

Table 2.210 – continued from previous page

weighted_inner(weight)	Weighted inner product on *Fn* spaces as free function.
weighted_norm(weight[, exponent])	Weighted norm on *Fn* spaces as free function.

Cn

odl.space.ntuples.**Cn**(*size*, *dtype='complex128'*, ***kwargs*)
The complex vector space C^n with vector multiplication.

Parameterssize : positive `int`

The number of dimensions of the space

dtype : `object`

The data type of the storage array. Can be provided in any way the `numpy.dtype` function understands, most notably as built-in type, as one of NumPy's internal datatype objects or as string.

Only complex floating-point data types are allowed.

kwargs : {'weight', 'dist', 'norm', 'inner', 'dist_using_inner'}

See *Fn*

See also:

Fnn-tuples over a field $\mathbb{F}$ with arbitrary scalar data type

Rn

odl.space.ntuples.**Rn**(*size*, *dtype='float64'*, ***kwargs*)
The real vector space R^n with vector multiplication.

Parameterssize : positive `int`

The number of dimensions of the space

dtype : `object`

The data type of the storage array. Can be provided in any way the `numpy.dtype` function understands, most notably as built-in type, as one of NumPy's internal datatype objects or as string.

Only real floating-point data types are allowed.

kwargs : {'weight', 'dist', 'norm', 'inner', 'dist_using_inner'}

See *Fn*

See also:

Fnn-tuples over a field $\mathbb{F}$ with arbitrary scalar data type

weighted_dist

odl.space.ntuples.**weighted_dist**(*weight*, *exponent=2.0*, *use_inner=False*)
Weighted distance on *Fn* spaces as free function.

Parametersweight : scalar or array-like

Weight of the distance. A scalar is interpreted as a constant weight, a 1-dim. array as a weighting vector and a 2-dimensional array as a weighting matrix.

exponent : positive `float`

Exponent of the norm. If `weight` is a sparse matrix, only 1.0, 2.0 and `inf` are allowed.

use_inner : `bool`, optional

Calculate `dist` using the formula

$\|x - y\|^2 = \|x\|^2 + \|y\|^2 - 2\Re\langle x, y\rangle$.

This avoids the creation of new arrays and is thus faster for large arrays. On the downside, it will not evaluate to exactly zero for equal (but not identical) x and y.

Can only be used if `exponent` is 2.0.

Returnsdist : `callable`

Distance function with given weight. Constant weightings are applicable to spaces of any size, for arrays the sizes of the weighting and the space must match.

See also:

FnConstWeighting, *FnVectorWeighting*, *FnMatrixWeighting*

weighted_inner

odl.space.ntuples.**weighted_inner**(*weight*)

Weighted inner product on *Fn* spaces as free function.

Parametersweight : scalar or array-like

Weight of the inner product. A scalar is interpreted as a constant weight, a 1-dim. array as a weighting vector and a 2-dimensional array as a weighting matrix.

Returnsinner : `callable`

Inner product function with given weight. Constant weightings are applicable to spaces of any size, for arrays the sizes of the weighting and the space must match.

See also:

FnConstWeighting, *FnVectorWeighting*, *FnMatrixWeighting*

weighted_norm

odl.space.ntuples.**weighted_norm**(*weight*, *exponent=2.0*)

Weighted norm on *Fn* spaces as free function.

Parametersweight : scalar or array-like

Weight of the norm. A scalar is interpreted as a constant weight, a 1-dim. array as a weighting vector and a 2-dimensional array as a weighting matrix.

exponent : positive `float`

Exponent of the norm. If `weight` is a sparse matrix, only 1.0, 2.0 and `inf` are allowed.

Returnsnorm : `callable`

Norm function with given weight. Constant weightings are applicable to spaces of any size, for arrays the sizes of the weighting and the space must match.

See also:

FnConstWeighting, *FnVectorWeighting*, *FnMatrixWeighting*

space_utils

Utility functions for space implementations.

Functions

vector(array[, dtype, impl])	Create an n-tuples type vector from an array.

vector

odl.space.space_utils.**vector**(*array*, *dtype=None*, *impl='numpy'*)

Create an n-tuples type vector from an array.

Parametersarray : array-like

Array from which to create the vector. Scalars become one-dimensional vectors.

dtype : `object`, optional

Set the data type of the vector manually with this option. By default, the space type is inferred from the input data.

impl : {'numpy', 'cuda'}

Implementation backend for the vector

Returnsvec : *NtuplesBaseVector*

Vector created from the input array. Its concrete type depends on the provided arguments.

Notes

This is a convenience function and not intended for use in speed-critical algorithms. It creates a NumPy array first, hence especially CUDA vectors as input result in a large speed penalty.

Examples

```
>>> vector([1, 2, 3])  # No automatic cast to float
Fn(3, 'int').element([1, 2, 3])
>>> vector([1, 2, 3], dtype=float)
Rn(3).element([1.0, 2.0, 3.0])
>>> vector([1 + 1j, 2, 3 - 2j])
Cn(3).element([(1+1j), (2+0j), (3-2j)])
```

Non-scalar types are also supported:

```
>>> vector([u'Hello,', u' world!'])
Ntuples(2, '<U7').element([u'Hello,', u' world!'])
```

Scalars become a one-element vector:

```
    >>> vector(0.0)
    Rn(1).element([0.0])
```

2.4.7 tomo

Tomography specific features

Modules

backends

Backends for other libraries

Modules

stir_bindings

Backend for STIR: Software for Tomographic Reconstruction

Back and forward projectors for PET.

ForwardProjectorByBinWrapper and *BackProjectorByBinWrapper* are general objects of STIR projectors and back-projectors, these can be used to wrap a given projector.

stir_projector_from_file allows users a easy way to create a *ForwardProjectorByBinWrapper* by giving file paths to the required templates.

References See the STIR webpage for more information and the STIR doc for info on the STIR classes used here.

Classes

BackProjectorByBinWrapper(dom, ran, volume, ...)	The back projector using STIR.
ForwardProjectorByBinWrapper(dom, ran, ...)	A forward projector using STIR.
StirVerbosity(verbosity)	Set STIR verbosity to a fixed level

BackProjectorByBinWrapper

class odl.tomo.backends.stir_bindings.**BackProjectorByBinWrapper**(*dom*, *ran*, *volume*, *proj_data*, *back_projector=None*, *adjoint=None*, *method=None*)

Bases: *odl.operator.operator.Operator*

The back projector using STIR.

Attributes

adjoint	The operator adjoint (abstract).

Continued on next page

Table 2.213 – continued from previous page

domain	The domain of this operator.
inverse	Return the operator inverse.
is_functional	`True` if the this operator's range is a *Field*.
is_linear	`True` if this operator is linear.
range	The range of this operator.

BackProjectorByBinWrapper.adjoint

`BackProjectorByBinWrapper.`**`adjoint`**

The operator adjoint (abstract).

RaisesOpNotImplementedError

Since the adjoint cannot be default implemented.

BackProjectorByBinWrapper.domain

`BackProjectorByBinWrapper.`**`domain`**

The domain of this operator.

BackProjectorByBinWrapper.inverse

`BackProjectorByBinWrapper.`**`inverse`**

Return the operator inverse.

RaisesOpNotImplementedError

Since the inverse cannot be default implemented.

BackProjectorByBinWrapper.is_functional

`BackProjectorByBinWrapper.`**`is_functional`**

`True` if the this operator's range is a *Field*.

BackProjectorByBinWrapper.is_linear

`BackProjectorByBinWrapper.`**`is_linear`**

`True` if this operator is linear.

BackProjectorByBinWrapper.range

`BackProjectorByBinWrapper.`**`range`**

The range of this operator.

Methods

`__eq__`	Return self==value.
_apply(projections, out)	Back project.

BackProjectorByBinWrapper._apply

`BackProjectorByBinWrapper.`**`_apply`**(*projections*, *out*)

Back project.

`__init__`(*dom*, *ran*, *volume*, *proj_data*, *back_projector=None*, *adjoint=None*, *method=None*)

Initialize a new back-projector.

Parametersdom : *DiscreteLp*

Projection space. Needs to have the same shape as `proj_data.to_array().shape()`.

ran : *DiscreteLp*

Volume of the projection. Needs to have the same shape as `volume.shape()`.

volume : `stir.FloatVoxelsOnCartesianGrid`

The stir volume to use in the forward projection

projection_data : `stir.ProjData`

The stir description of the projection.

back_projector : `stir.BackProjectorByBin`, optional

A pre-initialized back-projector.

adjoint : *ForwardProjectorByBinWrapper*, optional

A pre-initialized adjoint.

References

See STIR doc for info on the STIR classes.

ForwardProjectorByBinWrapper

class `odl.tomo.backends.stir_bindings.`**`ForwardProjectorByBinWrapper`**(*dom*, *ran*, *volume*, *proj_data*, *projector=None*, *adjoint=None*)

Bases: *odl.operator.operator.Operator*

A forward projector using STIR.

Uses "ForwardProjectorByBinUsingProjMatrixByBin" as a projector.

Attributes

adjoint	The back-projector associated with this operator.
domain	The domain of this operator.
inverse	Return the operator inverse.
is_functional	`True` if the this operator's range is a *Field*.
is_linear	`True` if this operator is linear.
range	The range of this operator.

ForwardProjectorByBinWrapper.adjoint

`ForwardProjectorByBinWrapper.`**`adjoint`**

The back-projector associated with this operator.

ForwardProjectorByBinWrapper.domain
`ForwardProjectorByBinWrapper.`**`domain`**
The domain of this operator.

ForwardProjectorByBinWrapper.inverse
`ForwardProjectorByBinWrapper.`**`inverse`**
Return the operator inverse.

RaisesOpNotImplementedError

Since the inverse cannot be default implemented.

ForwardProjectorByBinWrapper.is_functional
`ForwardProjectorByBinWrapper.`**`is_functional`**
`True` if the this operator's range is a *Field*.

ForwardProjectorByBinWrapper.is_linear
`ForwardProjectorByBinWrapper.`**`is_linear`**
`True` if this operator is linear.

ForwardProjectorByBinWrapper.range
`ForwardProjectorByBinWrapper.`**`range`**
The range of this operator.

Methods

`__eq__`	Return self==value.
_apply(volume, out)	Forward project a volume.

ForwardProjectorByBinWrapper._apply
`ForwardProjectorByBinWrapper.`**`_apply`**(*volume*, *out*)
Forward project a volume.
`__init__`(*dom*, *ran*, *volume*, *proj_data*, *projector=None*, *adjoint=None*)
Initialize a new projector.

Parametersdom : *DiscreteLp*

Volume of the projection. Needs to have the same shape as `volume.shape()`.

ran : *DiscreteLp*

Projection space. Needs to have the same shape as `proj_data.to_array().shape()`.

volume : `stir.FloatVoxelsOnCartesianGrid`

The stir volume to use in the forward projection

proj_data : `stir.ProjData`

The stir description of the projection.

projector : `stir.ForwardProjectorByBin`, optional

A pre-initialized projector.

adjoint : *BackProjectorByBinWrapper*, optional

A pre-initialized adjoint.

StirVerbosity

class odl.tomo.backends.stir_bindings.**StirVerbosity**(*verbosity*)

Bases: object

Set STIR verbosity to a fixed level

Methods

__eq__	Return self==value.

__init__(*verbosity*)

Functions

stir_projector_from_file(volume_file, ...)	Create a STIR projector from given template files.

stir_projector_from_file

odl.tomo.backends.stir_bindings.**stir_projector_from_file**(*volume_file*, *projection_file*)

Create a STIR projector from given template files.

Parametersvolume_file : str

Full file path to the STIR input file containing information on the volume. This is usually a '.hv' file. For STIR reasons, a '.v' file is also needed.

projection_file : str

Full file path to the STIR input file with information on the projection data. This is usually a '.hs' file. For STIR reasons, a '.s' file is also needed.

Returnsprojector : *ForwardProjectorByBinWrapper*

A STIR forward projector.

2.4.8 trafos

Function transformations based on ODL.

Modules

wavelet

Discrete wavelet transformation on L2 spaces.

Classes

DiscreteWaveletTransform(dom, nscales, ...)	Discrete wavelet trafo between discrete L2 spaces.
DiscreteWaveletTransformInverse(ran, ...)	Discrete inverse wavelet trafo between discrete L2 spaces.

DiscreteWaveletTransform

class odl.trafos.wavelet.**DiscreteWaveletTransform**(*dom*, *nscales*, *wbasis*, *mode*)

Bases: *odl.operator.operator.Operator*

Discrete wavelet trafo between discrete L2 spaces.

Attributes

adjoint	The adjoint wavelet transform.
domain	The domain of this operator.
inverse	The inverse wavelet transform.
is_biorthogonal	Whether or not the wavelet basis is bi-orthogonal.
is_functional	`True` if the this operator's range is a *Field*.
is_linear	`True` if this operator is linear.
is_orthogonal	Whether or not the wavelet basis is orthogonal.
range	The range of this operator.

DiscreteWaveletTransform.adjoint

DiscreteWaveletTransform.**adjoint**

The adjoint wavelet transform.

DiscreteWaveletTransform.domain

DiscreteWaveletTransform.**domain**

The domain of this operator.

DiscreteWaveletTransform.inverse

DiscreteWaveletTransform.**inverse**

The inverse wavelet transform.

DiscreteWaveletTransform.is_biorthogonal

DiscreteWaveletTransform.**is_biorthogonal**

Whether or not the wavelet basis is bi-orthogonal.

DiscreteWaveletTransform.is_functional

DiscreteWaveletTransform.**is_functional**

`True` if the this operator's range is a *Field*.

DiscreteWaveletTransform.is_linear

DiscreteWaveletTransform.**is_linear**

`True` if this operator is linear.

DiscreteWaveletTransform.is_orthogonal
`DiscreteWaveletTransform.`**`is_orthogonal`**
Whether or not the wavelet basis is orthogonal.

DiscreteWaveletTransform.range
`DiscreteWaveletTransform.`**`range`**
The range of this operator.

Methods

`__eq__`	Return self==value.
`_call`(x)	Compute the discrete 1D, 2D and 3D multiresolution wavelet transform

DiscreteWaveletTransform._call
`DiscreteWaveletTransform.`**`_call`**(*x*)
Compute the discrete 1D, 2D and 3D multiresolution wavelet transform Parameters ———- x : `DiscreteLp.Vector`

Returnsarr : `numpy.ndarray`

Flattened and concatenated coefficient array The length of the array depends on the size of input image to be transformed and on the chosen wavelet basis.

`__init__`(*dom*, *nscales*, *wbasis*, *mode*)
Initialize a new instance.

Parametersdom : *`DiscreteLp`*

Domain of the wavelet transform (the "image domain"). The exponent p of the discrete L^p space must be equal to 2.0.

nscales : `int`

Number of scales in the coefficient list

wbasis : `_pywt.Wavelet`

Describes properties of a selected wavelet basis

mode : `str`

Signal extension mode. For possible extensions see `pywt.MODES.modes` http://www.pybytes.com/pywavelets/ref/signal-extension-modes.html

DiscreteWaveletTransformInverse

class `odl.trafos.wavelet.`**`DiscreteWaveletTransformInverse`**(*ran*, *nscales*, *wbasis*, *mode*)
Bases: *`odl.operator.operator.Operator`*

Discrete inverse wavelet trafo between discrete L2 spaces.

Attributes

`adjoint`	The adjoint wavelet transform.

Continued on next page

Table 2.222 – continued from previous page

domain	The domain of this operator.
inverse	The inverse wavelet transform.
is_biorthogonal	Whether or not the wavelet basis is bi-orthogonal.
is_functional	`True` if the this operator's range is a *Field*.
is_linear	`True` if this operator is linear.
is_orthogonal	Whether or not the wavelet basis is orthogonal.
range	The range of this operator.

DiscreteWaveletTransformInverse.adjoint

`DiscreteWaveletTransformInverse.`**`adjoint`**

The adjoint wavelet transform.

DiscreteWaveletTransformInverse.domain

`DiscreteWaveletTransformInverse.`**`domain`**

The domain of this operator.

DiscreteWaveletTransformInverse.inverse

`DiscreteWaveletTransformInverse.`**`inverse`**

The inverse wavelet transform.

DiscreteWaveletTransformInverse.is_biorthogonal

`DiscreteWaveletTransformInverse.`**`is_biorthogonal`**

Whether or not the wavelet basis is bi-orthogonal.

DiscreteWaveletTransformInverse.is_functional

`DiscreteWaveletTransformInverse.`**`is_functional`**

`True` if the this operator's range is a *Field*.

DiscreteWaveletTransformInverse.is_linear

`DiscreteWaveletTransformInverse.`**`is_linear`**

`True` if this operator is linear.

DiscreteWaveletTransformInverse.is_orthogonal

`DiscreteWaveletTransformInverse.`**`is_orthogonal`**

Whether or not the wavelet basis is orthogonal.

DiscreteWaveletTransformInverse.range

`DiscreteWaveletTransformInverse.`**`range`**

The range of this operator.

Methods

`__eq__`	Return self==value.
_call(coeff)	Compute the discrete 1D, 2D or 3D inverse wavelet transform.

DiscreteWaveletTransformInverse._call

`DiscreteWaveletTransformInverse.`**`_call`**(*coeff*)

Compute the discrete 1D, 2D or 3D inverse wavelet transform.

Parameters
coeff : `DiscreteLp.Vector`

Returns
arr : `DiscreteLp.Vector`

`__init__`(*ran*, *nscales*, *wbasis*, *mode*)

Initialize a new instance.

Parameters
ran : *DiscreteLp*

Domain of the wavelet transform (the "image domain"). The exponent `p` of the discrete L^p space must be equal to 2.0.

nscales : `int`

Number of scales in the coefficient list

wbasis : `_pywt.Wavelet`

Describes properties of a selected wavelet basis

mode : `str`

Signal extension mode. For possible extensions see `pywt.MODES.modes` http://www.pybytes.com/pywavelets/ref/signal-extension-modes.html

Functions

array_to_pywt_coeff(coeff, size_list)	Convert a flat array into a pywt coefficient list.
coeff_size_list(shape, nscales, wbasis, mode)	Construct a size list from given wavelet coefficients.
pywt_coeff_to_array(coeff, size_list)	Convert a pywt coefficient into a flat array.
wavelet_decomposition3d(x, wbasis, mode, nscales)	Discrete 3D multiresolution wavelet decomposition.
wavelet_reconstruction3d(coeff_list, wbasis, ...)	Discrete 3D multiresolution wavelet reconstruction

array_to_pywt_coeff

`odl.trafos.wavelet.`**`array_to_pywt_coeff`**(*coeff*, *size_list*)

Convert a flat array into a pywt coefficient list.

For multilevel 1D, 2D and 3D discrete wavelet transforms.

Parameters
coeff : `DiscreteLp Vector`

A flat coefficient vector containing the approximation, and detail coefficients in the following order [aaaN, aadN, adaN, addN, daaN, dadN, ddaN, dddN, ... aad1, ada1, add1, daa1, dad1, dda1, ddd1]

size_list : list

A list of coefficient sizes such that,

`size_list[0]` = size of approximation coefficients at the coarsestlevel,

`size_list[1]` = size of the detailedetails at the coarsest level,

`size_list[N]` = size of the detailed coefficients at the finestlevel,

`size_list[N+1]` = size of original image,

`N` = the number of scaling levels

Returnscoeff : ordered `list`

Coefficient are organized in the list in the following way:

In 1D:

```
[aN, (dN), ...  (d1)]
```

The abbreviations refer to

`a` = approximation,

`d` = detail,

In 2D:

```
[aaN, (adN, daN, ddN), ...  (ad1, da1, dd1)]
```

The abbreviations refer to

`aa` = approx. on 1st dim, approx. on 2nd dim (approximation),

`ad` = approx. on 1st dim, detail on 2nd dim (horizontal),

`da` = detail on 1st dim, approx. on 2nd dim (vertical),

`dd` = detail on 1st dim, detail on 2nd dim (diagonal),

In 3D:

```
[aaaN, (aadN, adaN, addN, daaN, dadN, ddaN, dddN), ...
(aad1, ada1, add1, daa1, dad1, dda1, ddd1)]
```

The appreviations refer to,

`aaa` = approx. on 1st dim, approx. on 2nd dim, approx. on 3rd dim,

`aad` = approx. on 1st dim, approx. on 2nd dim, detail on 3rd dim,

`ada` = approx. on 1st dim, detail on 3nd dim, approx. on 3rd dim,

`add` = approx. on 1st dim, detail on 3nd dim, detail on 3rd dim,

`daa` = detail on 1st dim, approx. on 2nd dim, approx. on 3rd dim,

`dad` = detail on 1st dim, approx. on 2nd dim, detail on 3rd dim,

`dda` = detail on 1st dim, detail on 2nd dim, approx. on 3rd dim,

`ddd` = detail on 1st dim, detail on 2nd dim, detail on 3rd dim,

`N` refers to the number of scaling levels

coeff_size_list

`odl.trafos.wavelet.`**`coeff_size_list`**(*shape*, *nscales*, *wbasis*, *mode*)

Construct a size list from given wavelet coefficients.

Related to 1D, 2D and 3D multidimensional wavelet transforms that utilize PyWavelets.

Parametersshape : `tuple`

Number of pixels/voxels in the image. Its length must be 1, 2 or 3.

nscales : `int`

Number of scales in the multidimensional wavelet transform. This parameter is checked against the maximum number of scales returned by `pywt.dwt_max_level`. For more information see the corresponding documentation of PyWavelets.

wbasis : `pywt.Wavelet`

Selected wavelet basis

mode : `str`

Signal extention mode. Possible extension modes are

'zpd': zero-padding – signal is extended by adding zero samples

'cpd': constant padding – border values are replicated

'sym': symmetric padding – signal extension by mirroring samples

'ppd': periodic padding – signal is trated as a periodic one

'sp1': smooth padding – signal is extended according to the first derivatives calculated on the edges (straight line)

'per': periodization – like periodic-padding but gives the smallest possible number of decomposition coefficients.

Returnssize_list : `list`

A list containing the sizes of the wavelet (approximation and detail) coefficients at different scaling levels:

`size_list[0]` = size of approximation coefficients at the coarsest level

`size_list[1]` = size of the detail coefficients at the coarsest level

...

`size_list[N]` = size of the detail coefficients at the finest level

`size_list[N+1]` = size of the original image

`N` = number of scaling levels = nscales

pywt_coeff_to_array

`odl.trafos.wavelet.`**`pywt_coeff_to_array`**(*coeff*, *size_list*)

Convert a pywt coefficient into a flat array.

Related to 1D, 2D and 3D multilevel discrete wavelet transforms.

Parameterscoeff : ordered `list`

Coefficient are organized in the list in the following way:

In 1D:

```
[aN, (dN), ...  (d1)]
```

The abbreviations refer to

`a` = approximation,

`d` = detail

In 2D:

```
[aaN, (adN, daN, ddN), ...  (ad1, da1, dd1)]
```

The abbreviations refer to

`aa` = approx. on 1st dim, approx. on 2nd dim (approximation),

`ad` = approx. on 1st dim, detail on 2nd dim (horizontal),

`da` = detail on 1st dim, approx. on 2nd dim (vertical),

`dd` = detail on 1st dim, detail on 2nd dim (diagonal),

In 3D:

```
[aaaN, (aadN, adaN, addN, daaN, dadN, ddaN, dddN), ...
(aad1, ada1, add1, daa1, dad1, dda1, ddd1)]
```

The appreviations refer to

`aaa` = approx. on 1st dim, approx. on 2nd dim, approx. on 3rd dim,

`aad` = approx. on 1st dim, approx. on 2nd dim, detail on 3rd dim,

`ada` = approx. on 1st dim, detail on 3nd dim, approx. on 3rd dim,

`add` = approx. on 1st dim, detail on 3nd dim, detail on 3rd dim,

`daa` = detail on 1st dim, approx. on 2nd dim, approx. on 3rd dim,

`dad` = detail on 1st dim, approx. on 2nd dim, detail on 3rd dim,

`dda` = detail on 1st dim, detail on 2nd dim, approx. on 3rd dim,

`ddd` = detail on 1st dim, detail on 2nd dim, detail on 3rd dim,

`N` refers to the number of scaling levels

size_list : `list`

A list containing the sizes of the wavelet (approximation and detail) coefficients at different scaling levels

`size_list[0]` = size of approximation coefficients atthe coarsest level,

`size_list[1]` = size of the detailed coefficients atthe coarsest level,

`size_list[N]` = size of the detailed coefficients atthe finest level,

`size_list[N+1]` = size of original image, `N` = the number of scaling levels

Returnsarr : `numpy.ndarray`

Flattened and concatenated coefficient array The length of the array depends on the size of input image to be transformed and on the chosen wavelet basis.

wavelet_decomposition3d

`odl.trafos.wavelet.`**`wavelet_decomposition3d`**(*x*, *wbasis*, *mode*, *nscales*)

Discrete 3D multiresolution wavelet decomposition.

Compute the discrete 3D multiresolution wavelet decomposition at the given level (nscales) for a given 3D image. Utilizes a n-dimensional PyWavelet function `pywt.dwtn`.

Parametersx : `DiscreteLp.Vector`

wbasis: ‘‘_pywt.Wavelet‘‘

Describes properties of a selected wavelet basis

mode : `str`

Signal extention mode. For possible extensions see the signal extenstion modes of PyWavelets.

nscales : `int`

Number of scales in the coefficient list

Returnscoeff_list : `list`

A list of coefficient organized in the following way `'[aaaN, (aadN, adaN, addN, daaN, dadN, ddaN, dddN), ... (aad1, ada1, add1, daa1, dad1, dda1, ddd1)]'`.

The appreviations refer to,

`aaa` = approx. on 1st dim, approx. on 2nd dim, approx. on 3rd dim,

`aad` = approx. on 1st dim, approx. on 2nd dim, detail on 3rd dim,

`ada` = approx. on 1st dim, detail on 3nd dim, approx. on 3rd dim,

`add` = approx. on 1st dim, detail on 3nd dim, detail on 3rd dim,

`daa` = detail on 1st dim, approx. on 2nd dim, approx. on 3rd dim,

`dad` = detail on 1st dim, approx. on 2nd dim, detail on 3rd dim,

`dda` = detail on 1st dim, detail on 2nd dim, approx. on 3rd dim,

`ddd` = detail on 1st dim, detail on 2nd dim, detail on 3rd dim,

`N` = the number of scaling levels

wavelet_reconstruction3d

`odl.trafos.wavelet.`**`wavelet_reconstruction3d`**(*coeff_list*, *wbasis*, *mode*, *nscales*)

Discrete 3D multiresolution wavelet reconstruction

Compute a discrete 3D multiresolution wavelet reconstruction from a given wavelet coefficient list. Utilizes a PyWavelet function `pywt.dwtn`

Parameterscoeff_list : `list`

A list of wavelet approximation and detail coefficients organized in the following way `'[caaaN, (aadN, adaN, addN, daaN, dadN, ddaN, dddN), ... (aad1, ada1, add1, daa1, dad1, dda1, ddd1)]'`.

The appreviations refer to

`aaa` = approx. on 1st dim, approx. on 2nd dim, approx. on 3rd dim,

`aad` = approx. on 1st dim, approx. on 2nd dim, detail on3rd dim,

`ada` = approx. on 1st dim, detail on 3nd dim, approx. on 3rd dim,

`add` = approx. on 1st dim, detail on 3nd dim, detail on 3rd dim,

`daa` = detail on 1st dim, approx. on 2nd dim, approx. on 3rd dim,

`dad` = detail on 1st dim, approx. on 2nd dim, detail on 3rd dim,

`dda` = detail on 1st dim, detail on 2nd dim, approx. on 3rd dim,

`ddd` = detail on 1st dim, detail on 2nd dim, detail on 3rd dim,

N = the number of scaling levels

wbasis : `_pywt.Wavelet`

Describes properties of a selected wavelet basisodl.discr.lp_discr.

mode : `str`

Signal extention mode. For possible extensions see the signal extenstion modes of PyWavelets.

nscales : `int`

Number of scales in the coefficient list

Returnsx : `numpy.ndarray`.

A wavalet reconstruction.

2.4.9 util

Utility library for ODL, only for internal use.

Modules

graphics

Functions for graphical output.

Functions

show_discrete_function(dfunc[, method, ...])	Display a discrete 1d or 2d function.

show_discrete_function

odl.util.graphics.**show_discrete_function**(*dfunc*, *method=''*, *title=None*, *indices=None*, ***kwargs*)

Display a discrete 1d or 2d function.

Parametersdfunc : *DiscreteLpVector*

The discretized funciton to visualize.

method : `str`, optional

1d methods:

'plot' : graph plot

2d methods:

'imshow' : image plot with coloring according to value, including a colorbar.

'scatter' : cloud of scattered 3d points (3rd axis <-> value)

'wireframe', 'plot_wireframe' : surface plot

title : `str`, optional

Set the title of the figure

indices : index expression, optional

Display a slice of the array instead of the full array. The index expression is most easily created with the `numpy.s_` constructur, i.e. supply `np.s_[:, 1, :]` to display the first slice along the second axis.

For data with 3 or more dimensions, the 2d slice in the first two axes at the "middle" along the remaining axes is shown (semantically `[:, :, shape[2:] // 2]`).

kwargs : {'figsize', 'saveto', ...}

Extra keyword arguments passed on to display method See the Matplotlib functions for documentation of extra options.

See also:

matplotlib.pyplot.plot Show graph plot

matplotlib.pyplot.imshow Show data as image

matplotlib.pyplot.scatter Show scattered 3d points

phantom

Utilities for internal use.

Functions

derenzo_sources(space)	Creates the PET/SPECT derenzo phantom.
shepp_logan(space[, modified])	Create a Shepp-Logan phantom.

derenzo_sources

odl.util.phantom.**derenzo_sources**(*space*)

Creates the PET/SPECT derenzo phantom.

The derenzo phantom contains a series of circles of decreasing size.

shepp_logan

odl.util.phantom.**shepp_logan**(*space*, *modified=True*)

Create a Shepp-Logan phantom.

The shepp-logan phantom

References

Wikipedia : https://en.wikipedia.org/wiki/Shepp%E2%80%93Logan_phantom

testutils

Utilities for internal use.

Classes

FailCounter([err_msg])	Used to count the number of failures of something
ProgressBar([text])	A simple command-line progress bar.
ProgressRange(text, n)	**Methods**
Timer([name])	A timer context manager.

FailCounter

class odl.util.testutils.**FailCounter**(*err_msg=None*)
Bases: object

Used to count the number of failures of something

Useage:

```
with FailCounter() as counter:
    # Do stuff

    counter.fail()
```

When done, it prints

```
*** FAILED 1 TEST CASE(S) ***
```

Methods

__eq__	Return self==value.
fail([string])	

FailCounter.fail
FailCounter.**fail**(*string=None*)
__init__(*err_msg=None*)

ProgressBar

class odl.util.testutils.**ProgressBar**(*text='progress'*, **njobs*)
Bases: object

A simple command-line progress bar.

Usage:

```
>>> progress = ProgressBar('Reading data', 10)
```

Reading data: [ ] Starting

```
>>> progress.update(4) #halfway, zero indexing
```

Reading data: [###############] 50.0%

Also supports multiple index, from slowest varying to fastest

```
>>> progress = ProgressBar('Reading data', 10, 10)
```

Reading data: [ ] Starting

```
>>> progress.update(9, 8)
```

Reading data: [##############################] 99.0%

Supports simply calling update, which moves the counter forward:

```
>>> progress = ProgressBar('Reading data', 10, 10)
```

Reading data: [ ] Starting

```
>>> progress.update()
```

Reading data: [ ] 1.0%

Methods

`__eq__`	Return self==value.
`start`()	
`update`(*indices)	

ProgressBar.start
`ProgressBar.`**`start`**`()`

ProgressBar.update
`ProgressBar.`**`update`**`(`**indices*`)`
`__init__``(`*text='progress'*, **njobs*`)`

ProgressRange

class `odl.util.testutils.`**`ProgressRange`**`(`*text*, *n*`)`
Bases: `object`

Methods

`__eq__`	Return self==value.

`__init__``(`*text*, *n*`)`

Timer

class `odl.util.testutils.`**`Timer`**`(`*name=None*`)`
Bases: `object`

A timer context manager.

Usage:

```
with Timer('name'):
    # Do stuff
```

Prints the time stuff took to execute.

Methods

`__eq__`	Return self==value.

__init__(*name=None*)

Functions

all_almost_equal(iter1, iter2[, places])	
all_equal(iter1, iter2)	
almost_equal(a, b[, places])	
is_subdict(subdict, dictionary)	
skip_if_no_cuda(function)	
skip_if_no_pywavelets(function)	
timeit(arg)	A timer decorator.

all_almost_equal

odl.util.testutils.**all_almost_equal**(*iter1*, *iter2*, *places=None*)

all_equal

odl.util.testutils.**all_equal**(*iter1*, *iter2*)

almost_equal

odl.util.testutils.**almost_equal**(*a*, *b*, *places=None*)

is_subdict

odl.util.testutils.**is_subdict**(*subdict*, *dictionary*)

skip_if_no_cuda

odl.util.testutils.**skip_if_no_cuda**(*function*)

skip_if_no_pywavelets

odl.util.testutils.**skip_if_no_pywavelets**(*function*)

timeit

odl.util.testutils.**timeit**(*arg*)

A timer decorator.

Usage:

```
@timeit
def myfunction(...):
    ...

@timeit('info string')
def myfunction(...):
    ...
```

ufuncs

UFuncs for ODL vectors.

These functions are internal and should only be used as methods on *NtuplesBaseVector* type spaces.

See numpy.ufuncs for more information.

Notes

The default implementation of these methods make heavy use of the `NtuplesBaseVector.__array__` to extract a `numpy.ndarray` from the vector, and then apply a ufunc to it. Afterwards, `NtuplesBaseVector.__array_wrap__` is used to re-wrap the data into the appropriate space.

Classes

CudaNtuplesUFuncs(vector)	**Methods**
DiscreteLpUFuncs(vector)	UFuncs for *DiscreteLpVector* objects.
NtuplesBaseUFuncs(vector)	UFuncs for *NtuplesBaseVector* objects.
NtuplesUFuncs(vector)	UFuncs for *NtuplesVector* objects.
ProductSpaceUFuncs(vector)	UFuncs for *ProductSpaceVector* objects.

CudaNtuplesUFuncs

class odl.util.ufuncs.**CudaNtuplesUFuncs**(*vector*)

Bases: *odl.util.ufuncs.NtuplesBaseUFuncs*

Methods

__eq__	Return self==value.
absolute([out])	Calculate the absolute value element-wise.
add(x2[, out])	Add arguments element-wise with numpy.
arccos([out])	Trigonometric inverse cosine, element-wise.
arccosh([out])	Inverse hyperbolic cosine, element-wise.

Continued on next page

Table 2.234 – continued from previous page

arcsin([out])	Inverse sine, element-wise.
arcsinh([out])	Inverse hyperbolic sine element-wise.
arctan([out])	Trigonometric inverse tangent, element-wise.
arctan2(x2[, out])	Element-wise arc tangent of x1/x2 choosing the quadrant correctly.
arctanh([out])	Inverse hyperbolic tangent element-wise.
bitwise_and(x2[, out])	Compute the bit-wise AND of two arrays element-wise.
bitwise_or(x2[, out])	Compute the bit-wise OR of two arrays element-wise.
bitwise_xor(x2[, out])	Compute the bit-wise XOR of two arrays element-wise.
ceil([out])	Return the ceiling of the input, element-wise.
conj([out])	Return the complex conjugate, element-wise.
copysign(x2[, out])	Change the sign of x1 to that of x2, element-wise.
cos([out])	Cosine element-wise.
cosh([out])	Hyperbolic cosine, element-wise.
deg2rad([out])	Convert angles from degrees to radians.
divide(x2[, out])	Divide arguments element-wise with numpy.
equal(x2[, out])	Return (x1 == x2) element-wise.
exp([out])	Calculate the exponential of all elements in the input array.
exp2([out])	Calculate 2**p for all p in the input array.
expm1([out])	Calculate exp(x) - 1 for all elements in the array.
floor([out])	Return the floor of the input, element-wise.
floor_divide(x2[, out])	Return the largest integer smaller or equal to the division of the inputs.
fmax(x2[, out])	Element-wise maximum of array elements.
fmin(x2[, out])	Element-wise minimum of array elements.
fmod(x2[, out])	Return the element-wise remainder of division.
greater(x2[, out])	Return the truth value of (x1 > x2) element-wise.
greater_equal(x2[, out])	Return the truth value of (x1 >= x2) element-wise.
hypot(x2[, out])	Given the “legs” of a right triangle, return its hypotenuse.
invert([out])	Compute bit-wise inversion, or bit-wise NOT, element-wise.
isfinite([out])	Test element-wise for finiteness (not infinity or not Not a Number).
isinf([out])	Test element-wise for positive or negative infinity.
isnan([out])	Test element-wise for NaN and return result as a boolean array.
left_shift(x2[, out])	Shift the bits of an integer to the left.
less(x2[, out])	Return the truth value of (x1 < x2) element-wise.
less_equal(x2[, out])	Return the truth value of (x1 =< x2) element-wise.
log([out])	Natural logarithm, element-wise.
log10([out])	Return the base 10 logarithm of the input array, element-wise.
log1p([out])	Return the natural logarithm of one plus the input array, element-wise.
log2([out])	Base-2 logarithm of x.
logaddexp(x2[, out])	Logarithm of the sum of exponentiations of the inputs.
logaddexp2(x2[, out])	Logarithm of the sum of exponentiations of the inputs in base-2.
logical_and(x2[, out])	Compute the truth value of x1 AND x2 element-wise.
logical_not([out])	Compute the truth value of NOT x element-wise.
logical_or(x2[, out])	Compute the truth value of x1 OR x2 element-wise.
logical_xor(x2[, out])	Compute the truth value of x1 XOR x2, element-wise.
max()	Maximum value in array.
maximum(x2[, out])	Element-wise maximum of array elements.
min()	Minimum value in array.
minimum(x2[, out])	Element-wise minimum of array elements.
mod(x2[, out])	Return element-wise remainder of division.
modf([out1, out2])	Return the fractional and integral parts of an array, element-wise.

Continued on next page

Table 2.234 – continued from previous page

multiply(x2[, out])	Multiply arguments element-wise with numpy.
negative([out])	Numerical negative, element-wise.
nextafter(x2[, out])	Return the next floating-point value after x1 towards x2, element-wise.
not_equal(x2[, out])	Return (x1 != x2) element-wise.
power(x2[, out])	First array elements raised to powers from second array, element-wise.
prod()	Product of array elements.
rad2deg([out])	Convert angles from radians to degrees.
reciprocal([out])	Return the reciprocal of the argument, element-wise.
remainder(x2[, out])	Return element-wise remainder of division.
right_shift(x2[, out])	Shift the bits of an integer to the right.
rint([out])	Round elements of the array to the nearest integer.
sign([out])	Returns an element-wise indication of the sign of a number.
signbit([out])	Returns element-wise True where signbit is set (less than zero).
sin([out])	Trigonometric sine, element-wise.
sinh([out])	Hyperbolic sine, element-wise.
sqrt([out])	Return the positive square-root of an array, element-wise.
square([out])	Return the element-wise square of the input.
subtract(x2[, out])	Subtract arguments, element-wise with numpy.
sum()	Sum of array elements.
tan([out])	Compute tangent element-wise.
tanh([out])	Compute hyperbolic tangent element-wise.
true_divide(x2[, out])	Returns a true division of the inputs, element-wise.
trunc([out])	Return the truncated value of the input, element-wise.

CudaNtuplesUFuncs.absolute

`CudaNtuplesUFuncs.`**`absolute`**(*out=None*)

Calculate the absolute value element-wise.

See also:

`numpy.absolute`

CudaNtuplesUFuncs.add

`CudaNtuplesUFuncs.`**`add`**(*x2*, *out=None*)

Add arguments element-wise with numpy.

See also:

`numpy.add`

CudaNtuplesUFuncs.arccos

`CudaNtuplesUFuncs.`**`arccos`**(*out=None*)

Trigonometric inverse cosine, element-wise.

See also:

`numpy.arccos`

CudaNtuplesUFuncs.arccosh

`CudaNtuplesUFuncs.`**`arccosh`**(*out=None*)

Inverse hyperbolic cosine, element-wise.

See also:

numpy.arccosh

CudaNtuplesUFuncs.arcsin
CudaNtuplesUFuncs.**arcsin**(*out=None*)
Inverse sine, element-wise.

See also:

numpy.arcsin

CudaNtuplesUFuncs.arcsinh
CudaNtuplesUFuncs.**arcsinh**(*out=None*)
Inverse hyperbolic sine element-wise.

See also:

numpy.arcsinh

CudaNtuplesUFuncs.arctan
CudaNtuplesUFuncs.**arctan**(*out=None*)
Trigonometric inverse tangent, element-wise.

See also:

numpy.arctan

CudaNtuplesUFuncs.arctan2
CudaNtuplesUFuncs.**arctan2**(*x2*, *out=None*)
Element-wise arc tangent of x1/x2 choosing the quadrant correctly.

See also:

numpy.arctan2

CudaNtuplesUFuncs.arctanh
CudaNtuplesUFuncs.**arctanh**(*out=None*)
Inverse hyperbolic tangent element-wise.

See also:

numpy.arctanh

CudaNtuplesUFuncs.bitwise_and
CudaNtuplesUFuncs.**bitwise_and**(*x2*, *out=None*)
Compute the bit-wise AND of two arrays element-wise.

See also:

numpy.bitwise_and

CudaNtuplesUFuncs.bitwise_or
CudaNtuplesUFuncs.**bitwise_or**(*x2*, *out=None*)
Compute the bit-wise OR of two arrays element-wise.

See also:

numpy.bitwise_or

CudaNtuplesUFuncs.bitwise_xor
CudaNtuplesUFuncs.**bitwise_xor**(*x2*, *out=None*)
Compute the bit-wise XOR of two arrays element-wise.

See also:

numpy.bitwise_xor

CudaNtuplesUFuncs.ceil
CudaNtuplesUFuncs.**ceil**(*out=None*)
Return the ceiling of the input, element-wise.

See also:

numpy.ceil

CudaNtuplesUFuncs.conj
CudaNtuplesUFuncs.**conj**(*out=None*)
Return the complex conjugate, element-wise.

See also:

numpy.conj

CudaNtuplesUFuncs.copysign
CudaNtuplesUFuncs.**copysign**(*x2*, *out=None*)
Change the sign of x1 to that of x2, element-wise.

See also:

numpy.copysign

CudaNtuplesUFuncs.cos
CudaNtuplesUFuncs.**cos**(*out=None*)
Cosine element-wise.

See also:

numpy.cos

CudaNtuplesUFuncs.cosh
CudaNtuplesUFuncs.**cosh**(*out=None*)
Hyperbolic cosine, element-wise.

See also:

numpy.cosh

CudaNtuplesUFuncs.deg2rad
CudaNtuplesUFuncs.**deg2rad**(*out=None*)
Convert angles from degrees to radians.

See also:

numpy.deg2rad

CudaNtuplesUFuncs.divide
CudaNtuplesUFuncs.**divide**(*x2*, *out=None*)
Divide arguments element-wise with numpy.

See also:

numpy.divide

CudaNtuplesUFuncs.equal
CudaNtuplesUFuncs.**equal**(*x2*, *out=None*)
Return (x1 == x2) element-wise.

See also:

numpy.equal

CudaNtuplesUFuncs.exp
CudaNtuplesUFuncs.**exp**(*out=None*)
Calculate the exponential of all elements in the input array.

See also:

numpy.exp

CudaNtuplesUFuncs.exp2
CudaNtuplesUFuncs.**exp2**(*out=None*)
Calculate 2**p for all p in the input array.

See also:

numpy.exp2

CudaNtuplesUFuncs.expm1
CudaNtuplesUFuncs.**expm1**(*out=None*)
Calculate exp(x) - 1 for all elements in the array.

See also:

numpy.expm1

CudaNtuplesUFuncs.floor
CudaNtuplesUFuncs.**floor**(*out=None*)
Return the floor of the input, element-wise.

See also:

numpy.floor

CudaNtuplesUFuncs.floor_divide
CudaNtuplesUFuncs.**floor_divide**(*x2*, *out=None*)
Return the largest integer smaller or equal to the division of the inputs.

See also:

numpy.floor_divide

CudaNtuplesUFuncs.fmax
CudaNtuplesUFuncs.**fmax**(*x2*, *out=None*)
Element-wise maximum of array elements.

See also:

numpy.fmax

CudaNtuplesUFuncs.fmin
CudaNtuplesUFuncs.**fmin**(*x2*, *out=None*)
Element-wise minimum of array elements.

See also:

numpy.fmin

CudaNtuplesUFuncs.fmod
CudaNtuplesUFuncs.**fmod**(*x2*, *out=None*)
Return the element-wise remainder of division.

See also:

numpy.fmod

CudaNtuplesUFuncs.greater
CudaNtuplesUFuncs.**greater**(*x2*, *out=None*)
Return the truth value of (x1 > x2) element-wise.

See also:

numpy.greater

CudaNtuplesUFuncs.greater_equal
CudaNtuplesUFuncs.**greater_equal**(*x2*, *out=None*)
Return the truth value of (x1 >= x2) element-wise.

See also:

numpy.greater_equal

CudaNtuplesUFuncs.hypot
CudaNtuplesUFuncs.**hypot**(*x2*, *out=None*)
Given the "legs" of a right triangle, return its hypotenuse.

See also:

numpy.hypot

CudaNtuplesUFuncs.invert
CudaNtuplesUFuncs.**invert**(*out=None*)
Compute bit-wise inversion, or bit-wise NOT, element-wise.

See also:

numpy.invert

CudaNtuplesUFuncs.isfinite
CudaNtuplesUFuncs.**isfinite**(*out=None*)
Test element-wise for finiteness (not infinity or not Not a Number).

See also:

numpy.isfinite

CudaNtuplesUFuncs.isinf
CudaNtuplesUFuncs.**isinf**(*out=None*)
Test element-wise for positive or negative infinity.

See also:

numpy.isinf

CudaNtuplesUFuncs.isnan
CudaNtuplesUFuncs.**isnan**(*out=None*)
Test element-wise for NaN and return result as a boolean array.

See also:

numpy.isnan

CudaNtuplesUFuncs.left_shift
CudaNtuplesUFuncs.**left_shift**(*x2*, *out=None*)
Shift the bits of an integer to the left.

See also:

numpy.left_shift

CudaNtuplesUFuncs.less
CudaNtuplesUFuncs.**less**(*x2*, *out=None*)
Return the truth value of (x1 < x2) element-wise.

See also:

numpy.less

CudaNtuplesUFuncs.less_equal
CudaNtuplesUFuncs.**less_equal**(*x2*, *out=None*)
Return the truth value of (x1 =< x2) element-wise.

See also:

numpy.less_equal

CudaNtuplesUFuncs.log
CudaNtuplesUFuncs.**log**(*out=None*)
Natural logarithm, element-wise.

See also:

numpy.log

CudaNtuplesUFuncs.log10
CudaNtuplesUFuncs.**log10**(*out=None*)
Return the base 10 logarithm of the input array, element-wise.

See also:

numpy.log10

CudaNtuplesUFuncs.log1p
CudaNtuplesUFuncs.**log1p**(*out=None*)
Return the natural logarithm of one plus the input array, element-wise.

See also:

numpy.log1p

CudaNtuplesUFuncs.log2
CudaNtuplesUFuncs.**log2**(*out=None*)
Base-2 logarithm of x.

See also:

numpy.log2

CudaNtuplesUFuncs.logaddexp
CudaNtuplesUFuncs.**logaddexp**(*x2*, *out=None*)
Logarithm of the sum of exponentiations of the inputs.

See also:

numpy.logaddexp

CudaNtuplesUFuncs.logaddexp2
CudaNtuplesUFuncs.**logaddexp2**(*x2*, *out=None*)
Logarithm of the sum of exponentiations of the inputs in base-2.

See also:

numpy.logaddexp2

CudaNtuplesUFuncs.logical_and
CudaNtuplesUFuncs.**logical_and**(*x2*, *out=None*)
Compute the truth value of x1 AND x2 element-wise.

See also:

numpy.logical_and

CudaNtuplesUFuncs.logical_not
CudaNtuplesUFuncs.**logical_not**(*out=None*)
Compute the truth value of NOT x element-wise.

See also:

numpy.logical_not

CudaNtuplesUFuncs.logical_or
CudaNtuplesUFuncs.**logical_or**(*x2*, *out=None*)
Compute the truth value of x1 OR x2 element-wise.

See also:

numpy.logical_or

CudaNtuplesUFuncs.logical_xor
CudaNtuplesUFuncs.**logical_xor**(*x2*, *out=None*)
Compute the truth value of x1 XOR x2, element-wise.

See also:

numpy.logical_xor

CudaNtuplesUFuncs.max
CudaNtuplesUFuncs.**max**()
Maximum value in array.

See also:

numpy.max

CudaNtuplesUFuncs.maximum
CudaNtuplesUFuncs.**maximum**(*x2*, *out=None*)
Element-wise maximum of array elements.

See also:

numpy.maximum

CudaNtuplesUFuncs.min
CudaNtuplesUFuncs.**min**()
Minimum value in array.

See also:

numpy.min

CudaNtuplesUFuncs.minimum
CudaNtuplesUFuncs.**minimum**(*x2*, *out=None*)
Element-wise minimum of array elements.

See also:

numpy.minimum

CudaNtuplesUFuncs.mod
CudaNtuplesUFuncs.**mod**(*x2*, *out=None*)
Return element-wise remainder of division.

See also:

numpy.mod

CudaNtuplesUFuncs.modf
CudaNtuplesUFuncs.**modf**(*out1=None*, *out2=None*)
Return the fractional and integral parts of an array, element-wise.

See also:

numpy.modf

CudaNtuplesUFuncs.multiply
CudaNtuplesUFuncs.**multiply**(*x2*, *out=None*)
Multiply arguments element-wise with numpy.

See also:

numpy.multiply

CudaNtuplesUFuncs.negative
CudaNtuplesUFuncs.**negative**(*out=None*)
Numerical negative, element-wise.

See also:

numpy.negative

CudaNtuplesUFuncs.nextafter
CudaNtuplesUFuncs.**nextafter**(*x2*, *out=None*)
Return the next floating-point value after x1 towards x2, element-wise.

See also:

numpy.nextafter

CudaNtuplesUFuncs.not_equal
CudaNtuplesUFuncs.**not_equal**(*x2*, *out=None*)
Return (x1 != x2) element-wise.

See also:

numpy.not_equal

CudaNtuplesUFuncs.power
CudaNtuplesUFuncs.**power**(*x2*, *out=None*)
First array elements raised to powers from second array, element-wise.

See also:

numpy.power

CudaNtuplesUFuncs.prod
CudaNtuplesUFuncs.**prod**()
Product of array elements.

See also:

numpy.prod

CudaNtuplesUFuncs.rad2deg
CudaNtuplesUFuncs.**rad2deg**(*out=None*)
Convert angles from radians to degrees.

See also:

numpy.rad2deg

CudaNtuplesUFuncs.reciprocal
CudaNtuplesUFuncs.**reciprocal**(*out=None*)
Return the reciprocal of the argument, element-wise.

See also:

numpy.reciprocal

CudaNtuplesUFuncs.remainder
CudaNtuplesUFuncs.**remainder**(*x2*, *out=None*)
Return element-wise remainder of division.

See also:

numpy.remainder

CudaNtuplesUFuncs.right_shift
CudaNtuplesUFuncs.**right_shift**(*x2*, *out=None*)
Shift the bits of an integer to the right.

See also:

numpy.right_shift

CudaNtuplesUFuncs.rint
CudaNtuplesUFuncs.**rint**(*out=None*)
Round elements of the array to the nearest integer.

See also:

numpy.rint

CudaNtuplesUFuncs.sign
CudaNtuplesUFuncs.**sign**(*out=None*)
Returns an element-wise indication of the sign of a number.

See also:

numpy.sign

CudaNtuplesUFuncs.signbit
CudaNtuplesUFuncs.**signbit**(*out=None*)
Returns element-wise True where signbit is set (less than zero).

See also:

numpy.signbit

CudaNtuplesUFuncs.sin
CudaNtuplesUFuncs.**sin**(*out=None*)
Trigonometric sine, element-wise.

See also:

numpy.sin

CudaNtuplesUFuncs.sinh
CudaNtuplesUFuncs.**sinh**(*out=None*)
Hyperbolic sine, element-wise.

See also:

numpy.sinh

CudaNtuplesUFuncs.sqrt
CudaNtuplesUFuncs.**sqrt**(*out=None*)
Return the positive square-root of an array, element-wise.

See also:

numpy.sqrt

CudaNtuplesUFuncs.square
CudaNtuplesUFuncs.**square**(*out=None*)
Return the element-wise square of the input.

See also:

numpy.square

CudaNtuplesUFuncs.subtract
CudaNtuplesUFuncs.**subtract**(*x2*, *out=None*)
Subtract arguments, element-wise with numpy.

See also:

numpy.subtract

CudaNtuplesUFuncs.sum
CudaNtuplesUFuncs.**sum**()
Sum of array elements.

See also:

numpy.sum

CudaNtuplesUFuncs.tan
CudaNtuplesUFuncs.**tan**(*out=None*)
Compute tangent element-wise.

See also:

numpy.tan

CudaNtuplesUFuncs.tanh
CudaNtuplesUFuncs.**tanh**(*out=None*)
Compute hyperbolic tangent element-wise.

See also:

numpy.tanh

CudaNtuplesUFuncs.true_divide
CudaNtuplesUFuncs.**true_divide**(*x2*, *out=None*)
Returns a true division of the inputs, element-wise.

See also:

numpy.true_divide

CudaNtuplesUFuncs.trunc
CudaNtuplesUFuncs.**trunc**(*out=None*)
Return the truncated value of the input, element-wise.

See also:

numpy.trunc
__init__(*vector*)
Create ufunc wrapper for vector.

DiscreteLpUFuncs
class odl.util.ufuncs.**DiscreteLpUFuncs**(*vector*)
Bases: *odl.util.ufuncs.NtuplesBaseUFuncs*

UFuncs for *DiscreteLpVector* objects.

Internal object, should not be created except in *DiscreteLpVector*.

Methods

__eq__	Return self==value.
absolute([out])	Calculate the absolute value element-wise.
add(x2[, out])	Add arguments element-wise with numpy.
arccos([out])	Trigonometric inverse cosine, element-wise.
arccosh([out])	Inverse hyperbolic cosine, element-wise.
arcsin([out])	Inverse sine, element-wise.
arcsinh([out])	Inverse hyperbolic sine element-wise.
arctan([out])	Trigonometric inverse tangent, element-wise.
arctan2(x2[, out])	Element-wise arc tangent of x1/x2 choosing the quadrant correctly.
arctanh([out])	Inverse hyperbolic tangent element-wise.
bitwise_and(x2[, out])	Compute the bit-wise AND of two arrays element-wise.
bitwise_or(x2[, out])	Compute the bit-wise OR of two arrays element-wise.
bitwise_xor(x2[, out])	Compute the bit-wise XOR of two arrays element-wise.
ceil([out])	Return the ceiling of the input, element-wise.
conj([out])	Return the complex conjugate, element-wise.
copysign(x2[, out])	Change the sign of x1 to that of x2, element-wise.
cos([out])	Cosine element-wise.

Continued on next page

Table 2.235 – continued from previous page

cosh([out])	Hyperbolic cosine, element-wise.
deg2rad([out])	Convert angles from degrees to radians.
divide(x2[, out])	Divide arguments element-wise with numpy.
equal(x2[, out])	Return (x1 == x2) element-wise.
exp([out])	Calculate the exponential of all elements in the input array.
exp2([out])	Calculate 2**p for all p in the input array.
expm1([out])	Calculate exp(x) - 1 for all elements in the array.
floor([out])	Return the floor of the input, element-wise.
floor_divide(x2[, out])	Return the largest integer smaller or equal to the division of the inputs.
fmax(x2[, out])	Element-wise maximum of array elements.
fmin(x2[, out])	Element-wise minimum of array elements.
fmod(x2[, out])	Return the element-wise remainder of division.
greater(x2[, out])	Return the truth value of (x1 > x2) element-wise.
greater_equal(x2[, out])	Return the truth value of (x1 >= x2) element-wise.
hypot(x2[, out])	Given the "legs" of a right triangle, return its hypotenuse.
invert([out])	Compute bit-wise inversion, or bit-wise NOT, element-wise.
isfinite([out])	Test element-wise for finiteness (not infinity or not Not a Number).
isinf([out])	Test element-wise for positive or negative infinity.
isnan([out])	Test element-wise for NaN and return result as a boolean array.
left_shift(x2[, out])	Shift the bits of an integer to the left.
less(x2[, out])	Return the truth value of (x1 < x2) element-wise.
less_equal(x2[, out])	Return the truth value of (x1 =< x2) element-wise.
log([out])	Natural logarithm, element-wise.
log10([out])	Return the base 10 logarithm of the input array, element-wise.
log1p([out])	Return the natural logarithm of one plus the input array, element-wise.
log2([out])	Base-2 logarithm of x.
logaddexp(x2[, out])	Logarithm of the sum of exponentiations of the inputs.
logaddexp2(x2[, out])	Logarithm of the sum of exponentiations of the inputs in base-2.
logical_and(x2[, out])	Compute the truth value of x1 AND x2 element-wise.
logical_not([out])	Compute the truth value of NOT x element-wise.
logical_or(x2[, out])	Compute the truth value of x1 OR x2 element-wise.
logical_xor(x2[, out])	Compute the truth value of x1 XOR x2, element-wise.
max()	Maximum value in array.
maximum(x2[, out])	Element-wise maximum of array elements.
min()	Minimum value in array.
minimum(x2[, out])	Element-wise minimum of array elements.
mod(x2[, out])	Return element-wise remainder of division.
modf([out1, out2])	Return the fractional and integral parts of an array, element-wise.
multiply(x2[, out])	Multiply arguments element-wise with numpy.
negative([out])	Numerical negative, element-wise.
nextafter(x2[, out])	Return the next floating-point value after x1 towards x2, element-wise.
not_equal(x2[, out])	Return (x1 != x2) element-wise.
power(x2[, out])	First array elements raised to powers from second array, element-wise.
prod()	Product of array elements.
rad2deg([out])	Convert angles from radians to degrees.
reciprocal([out])	Return the reciprocal of the argument, element-wise.
remainder(x2[, out])	Return element-wise remainder of division.
right_shift(x2[, out])	Shift the bits of an integer to the right.
rint([out])	Round elements of the array to the nearest integer.
sign([out])	Returns an element-wise indication of the sign of a number.

Continued on next page

Table 2.235 – continued from previous page

signbit([out])	Returns element-wise True where signbit is set (less than zero).
sin([out])	Trigonometric sine, element-wise.
sinh([out])	Hyperbolic sine, element-wise.
sqrt([out])	Return the positive square-root of an array, element-wise.
square([out])	Return the element-wise square of the input.
subtract(x2[, out])	Subtract arguments, element-wise with numpy.
sum()	Sum of array elements.
tan([out])	Compute tangent element-wise.
tanh([out])	Compute hyperbolic tangent element-wise.
true_divide(x2[, out])	Returns a true division of the inputs, element-wise.
trunc([out])	Return the truncated value of the input, element-wise.

DiscreteLpUFuncs.absolute

DiscreteLpUFuncs.**absolute**(*out=None*)

Calculate the absolute value element-wise.

See also:

numpy.absolute

DiscreteLpUFuncs.add

DiscreteLpUFuncs.**add**(*x2*, *out=None*)

Add arguments element-wise with numpy.

See also:

numpy.add

DiscreteLpUFuncs.arccos

DiscreteLpUFuncs.**arccos**(*out=None*)

Trigonometric inverse cosine, element-wise.

See also:

numpy.arccos

DiscreteLpUFuncs.arccosh

DiscreteLpUFuncs.**arccosh**(*out=None*)

Inverse hyperbolic cosine, element-wise.

See also:

numpy.arccosh

DiscreteLpUFuncs.arcsin

DiscreteLpUFuncs.**arcsin**(*out=None*)

Inverse sine, element-wise.

See also:

numpy.arcsin

DiscreteLpUFuncs.arcsinh
DiscreteLpUFuncs.**arcsinh**(*out=None*)
Inverse hyperbolic sine element-wise.

See also:

numpy.arcsinh

DiscreteLpUFuncs.arctan
DiscreteLpUFuncs.**arctan**(*out=None*)
Trigonometric inverse tangent, element-wise.

See also:

numpy.arctan

DiscreteLpUFuncs.arctan2
DiscreteLpUFuncs.**arctan2**(*x2*, *out=None*)
Element-wise arc tangent of x1/x2 choosing the quadrant correctly.

See also:

numpy.arctan2

DiscreteLpUFuncs.arctanh
DiscreteLpUFuncs.**arctanh**(*out=None*)
Inverse hyperbolic tangent element-wise.

See also:

numpy.arctanh

DiscreteLpUFuncs.bitwise_and
DiscreteLpUFuncs.**bitwise_and**(*x2*, *out=None*)
Compute the bit-wise AND of two arrays element-wise.

See also:

numpy.bitwise_and

DiscreteLpUFuncs.bitwise_or
DiscreteLpUFuncs.**bitwise_or**(*x2*, *out=None*)
Compute the bit-wise OR of two arrays element-wise.

See also:

numpy.bitwise_or

DiscreteLpUFuncs.bitwise_xor
DiscreteLpUFuncs.**bitwise_xor**(*x2*, *out=None*)
Compute the bit-wise XOR of two arrays element-wise.

See also:

numpy.bitwise_xor

DiscreteLpUFuncs.ceil
DiscreteLpUFuncs.**ceil**(*out=None*)
Return the ceiling of the input, element-wise.

See also:

numpy.ceil

DiscreteLpUFuncs.conj
DiscreteLpUFuncs.**conj**(*out=None*)
Return the complex conjugate, element-wise.

See also:

numpy.conj

DiscreteLpUFuncs.copysign
DiscreteLpUFuncs.**copysign**(*x2*, *out=None*)
Change the sign of x1 to that of x2, element-wise.

See also:

numpy.copysign

DiscreteLpUFuncs.cos
DiscreteLpUFuncs.**cos**(*out=None*)
Cosine element-wise.

See also:

numpy.cos

DiscreteLpUFuncs.cosh
DiscreteLpUFuncs.**cosh**(*out=None*)
Hyperbolic cosine, element-wise.

See also:

numpy.cosh

DiscreteLpUFuncs.deg2rad
DiscreteLpUFuncs.**deg2rad**(*out=None*)
Convert angles from degrees to radians.

See also:

numpy.deg2rad

DiscreteLpUFuncs.divide
DiscreteLpUFuncs.**divide**(*x2*, *out=None*)
Divide arguments element-wise with numpy.

See also:

numpy.divide

DiscreteLpUFuncs.equal
DiscreteLpUFuncs.**equal**(*x2*, *out=None*)
Return (x1 == x2) element-wise.

See also:

numpy.equal

DiscreteLpUFuncs.exp
DiscreteLpUFuncs.**exp**(*out=None*)
Calculate the exponential of all elements in the input array.

See also:

numpy.exp

DiscreteLpUFuncs.exp2
DiscreteLpUFuncs.**exp2**(*out=None*)
Calculate 2**p for all p in the input array.

See also:

numpy.exp2

DiscreteLpUFuncs.expm1
DiscreteLpUFuncs.**expm1**(*out=None*)
Calculate exp(x) - 1 for all elements in the array.

See also:

numpy.expm1

DiscreteLpUFuncs.floor
DiscreteLpUFuncs.**floor**(*out=None*)
Return the floor of the input, element-wise.

See also:

numpy.floor

DiscreteLpUFuncs.floor_divide
DiscreteLpUFuncs.**floor_divide**(*x2*, *out=None*)
Return the largest integer smaller or equal to the division of the inputs.

See also:

numpy.floor_divide

DiscreteLpUFuncs.fmax
DiscreteLpUFuncs.**fmax**(*x2*, *out=None*)
Element-wise maximum of array elements.

See also:

numpy.fmax

DiscreteLpUFuncs.fmin
DiscreteLpUFuncs.**fmin**(*x2*, *out=None*)
Element-wise minimum of array elements.

See also:

numpy.fmin

DiscreteLpUFuncs.fmod
DiscreteLpUFuncs.**fmod**(*x2*, *out=None*)
Return the element-wise remainder of division.

See also:

numpy.fmod

DiscreteLpUFuncs.greater
DiscreteLpUFuncs.**greater**(*x2*, *out=None*)
Return the truth value of (x1 > x2) element-wise.

See also:

numpy.greater

DiscreteLpUFuncs.greater_equal
DiscreteLpUFuncs.**greater_equal**(*x2*, *out=None*)
Return the truth value of (x1 >= x2) element-wise.

See also:

numpy.greater_equal

DiscreteLpUFuncs.hypot
DiscreteLpUFuncs.**hypot**(*x2*, *out=None*)
Given the "legs" of a right triangle, return its hypotenuse.

See also:

numpy.hypot

DiscreteLpUFuncs.invert
DiscreteLpUFuncs.**invert**(*out=None*)
Compute bit-wise inversion, or bit-wise NOT, element-wise.

See also:

numpy.invert

DiscreteLpUFuncs.isfinite
DiscreteLpUFuncs.**isfinite**(*out=None*)
Test element-wise for finiteness (not infinity or not Not a Number).

See also:

numpy.isfinite

DiscreteLpUFuncs.isinf
DiscreteLpUFuncs.**isinf**(*out=None*)
Test element-wise for positive or negative infinity.

See also:

numpy.isinf

DiscreteLpUFuncs.isnan
DiscreteLpUFuncs.**isnan**(*out=None*)
Test element-wise for NaN and return result as a boolean array.

See also:

numpy.isnan

DiscreteLpUFuncs.left_shift
DiscreteLpUFuncs.**left_shift**(*x2*, *out=None*)
Shift the bits of an integer to the left.

See also:

numpy.left_shift

DiscreteLpUFuncs.less
DiscreteLpUFuncs.**less**(*x2*, *out=None*)
Return the truth value of (x1 < x2) element-wise.

See also:

numpy.less

DiscreteLpUFuncs.less_equal
DiscreteLpUFuncs.**less_equal**(*x2*, *out=None*)
Return the truth value of (x1 =< x2) element-wise.

See also:

numpy.less_equal

DiscreteLpUFuncs.log
DiscreteLpUFuncs.**log**(*out=None*)
Natural logarithm, element-wise.

See also:

numpy.log

DiscreteLpUFuncs.log10
DiscreteLpUFuncs.**log10**(*out=None*)
Return the base 10 logarithm of the input array, element-wise.

See also:

numpy.log10

DiscreteLpUFuncs.log1p
DiscreteLpUFuncs.**log1p**(*out=None*)
Return the natural logarithm of one plus the input array, element-wise.

See also:

numpy.log1p

DiscreteLpUFuncs.log2
DiscreteLpUFuncs.**log2**(*out=None*)
Base-2 logarithm of x.

See also:

numpy.log2

DiscreteLpUFuncs.logaddexp
DiscreteLpUFuncs.**logaddexp**(*x2*, *out=None*)
Logarithm of the sum of exponentiations of the inputs.

See also:

numpy.logaddexp

DiscreteLpUFuncs.logaddexp2
DiscreteLpUFuncs.**logaddexp2**(*x2*, *out=None*)
Logarithm of the sum of exponentiations of the inputs in base-2.

See also:

numpy.logaddexp2

DiscreteLpUFuncs.logical_and
DiscreteLpUFuncs.**logical_and**(*x2*, *out=None*)
Compute the truth value of x1 AND x2 element-wise.

See also:

numpy.logical_and

DiscreteLpUFuncs.logical_not
DiscreteLpUFuncs.**logical_not**(*out=None*)
Compute the truth value of NOT x element-wise.

See also:

numpy.logical_not

DiscreteLpUFuncs.logical_or
DiscreteLpUFuncs.**logical_or**(*x2*, *out=None*)
Compute the truth value of x1 OR x2 element-wise.

See also:

numpy.logical_or

DiscreteLpUFuncs.logical_xor
DiscreteLpUFuncs.**logical_xor**(*x2*, *out=None*)
Compute the truth value of x1 XOR x2, element-wise.

See also:

numpy.logical_xor

DiscreteLpUFuncs.max
DiscreteLpUFuncs.**max**()
Maximum value in array.

See also:

numpy.max

DiscreteLpUFuncs.maximum
DiscreteLpUFuncs.**maximum**(*x2*, *out=None*)
Element-wise maximum of array elements.

See also:

numpy.maximum

DiscreteLpUFuncs.min
DiscreteLpUFuncs.**min**()
Minimum value in array.

See also:

numpy.min

DiscreteLpUFuncs.minimum
DiscreteLpUFuncs.**minimum**(*x2*, *out=None*)
Element-wise minimum of array elements.

See also:

numpy.minimum

DiscreteLpUFuncs.mod
DiscreteLpUFuncs.**mod**(*x2*, *out=None*)
Return element-wise remainder of division.

See also:

numpy.mod

DiscreteLpUFuncs.modf
DiscreteLpUFuncs.**modf**(*out1=None*, *out2=None*)
Return the fractional and integral parts of an array, element-wise.

See also:

numpy.modf

DiscreteLpUFuncs.multiply
DiscreteLpUFuncs.**multiply**(*x2*, *out=None*)
Multiply arguments element-wise with numpy.

See also:

numpy.multiply

DiscreteLpUFuncs.negative
DiscreteLpUFuncs.**negative**(*out=None*)
Numerical negative, element-wise.

See also:

numpy.negative

DiscreteLpUFuncs.nextafter
DiscreteLpUFuncs.**nextafter**(*x2*, *out=None*)
Return the next floating-point value after x1 towards x2, element-wise.

See also:

numpy.nextafter

DiscreteLpUFuncs.not_equal
DiscreteLpUFuncs.**not_equal**(*x2*, *out=None*)
Return (x1 != x2) element-wise.

See also:

numpy.not_equal

DiscreteLpUFuncs.power
DiscreteLpUFuncs.**power**(*x2*, *out=None*)
First array elements raised to powers from second array, element-wise.

See also:

numpy.power

DiscreteLpUFuncs.prod
DiscreteLpUFuncs.**prod**()
Product of array elements.

See also:

numpy.prod

DiscreteLpUFuncs.rad2deg
DiscreteLpUFuncs.**rad2deg**(*out=None*)
Convert angles from radians to degrees.

See also:

numpy.rad2deg

DiscreteLpUFuncs.reciprocal
DiscreteLpUFuncs.**reciprocal**(*out=None*)
Return the reciprocal of the argument, element-wise.

See also:

numpy.reciprocal

DiscreteLpUFuncs.remainder
DiscreteLpUFuncs.**remainder**(*x2*, *out=None*)
Return element-wise remainder of division.

See also:

numpy.remainder

DiscreteLpUFuncs.right_shift
DiscreteLpUFuncs.**right_shift**(*x2*, *out=None*)
Shift the bits of an integer to the right.

See also:

numpy.right_shift

DiscreteLpUFuncs.rint
DiscreteLpUFuncs.**rint**(*out=None*)
Round elements of the array to the nearest integer.

See also:

numpy.rint

DiscreteLpUFuncs.sign
DiscreteLpUFuncs.**sign**(*out=None*)
Returns an element-wise indication of the sign of a number.

See also:

numpy.sign

DiscreteLpUFuncs.signbit
DiscreteLpUFuncs.**signbit**(*out=None*)
Returns element-wise True where signbit is set (less than zero).

See also:

numpy.signbit

DiscreteLpUFuncs.sin
DiscreteLpUFuncs.**sin**(*out=None*)
Trigonometric sine, element-wise.

See also:

numpy.sin

DiscreteLpUFuncs.sinh
DiscreteLpUFuncs.**sinh**(*out=None*)
Hyperbolic sine, element-wise.

See also:

numpy.sinh

DiscreteLpUFuncs.sqrt
DiscreteLpUFuncs.**sqrt**(*out=None*)
Return the positive square-root of an array, element-wise.

See also:

numpy.sqrt

DiscreteLpUFuncs.square
DiscreteLpUFuncs.**square**(*out=None*)
Return the element-wise square of the input.

See also:

numpy.square

DiscreteLpUFuncs.subtract
DiscreteLpUFuncs.**subtract**(*x2*, *out=None*)
Subtract arguments, element-wise with numpy.

See also:

numpy.subtract

DiscreteLpUFuncs.sum
DiscreteLpUFuncs.**sum**()
Sum of array elements.

See also:

numpy.sum

DiscreteLpUFuncs.tan
DiscreteLpUFuncs.**tan**(*out=None*)
Compute tangent element-wise.

See also:

numpy.tan

DiscreteLpUFuncs.tanh
DiscreteLpUFuncs.**tanh**(*out=None*)
Compute hyperbolic tangent element-wise.

See also:

numpy.tanh

DiscreteLpUFuncs.true_divide
DiscreteLpUFuncs.**true_divide**(*x2*, *out=None*)
Returns a true division of the inputs, element-wise.

See also:

numpy.true_divide

DiscreteLpUFuncs.trunc
DiscreteLpUFuncs.**trunc**(*out=None*)
Return the truncated value of the input, element-wise.

See also:

numpy.trunc
__init__(*vector*)
Create ufunc wrapper for vector.

NtuplesBaseUFuncs
class odl.util.ufuncs.**NtuplesBaseUFuncs**(*vector*)
Bases: object

UFuncs for *NtuplesBaseVector* objects.

Internal object, should not be created except in *NtuplesBaseVector*.

Methods

__eq__	Return self==value.
absolute([out])	Calculate the absolute value element-wise.
add(x2[, out])	Add arguments element-wise with numpy.
arccos([out])	Trigonometric inverse cosine, element-wise.
arccosh([out])	Inverse hyperbolic cosine, element-wise.
arcsin([out])	Inverse sine, element-wise.
arcsinh([out])	Inverse hyperbolic sine element-wise.
arctan([out])	Trigonometric inverse tangent, element-wise.
arctan2(x2[, out])	Element-wise arc tangent of x1/x2 choosing the quadrant correctly.
arctanh([out])	Inverse hyperbolic tangent element-wise.
bitwise_and(x2[, out])	Compute the bit-wise AND of two arrays element-wise.
bitwise_or(x2[, out])	Compute the bit-wise OR of two arrays element-wise.
bitwise_xor(x2[, out])	Compute the bit-wise XOR of two arrays element-wise.
ceil([out])	Return the ceiling of the input, element-wise.
conj([out])	Return the complex conjugate, element-wise.
copysign(x2[, out])	Change the sign of x1 to that of x2, element-wise.
cos([out])	Cosine element-wise.
cosh([out])	Hyperbolic cosine, element-wise.
deg2rad([out])	Convert angles from degrees to radians.
divide(x2[, out])	Divide arguments element-wise with numpy.
equal(x2[, out])	Return (x1 == x2) element-wise.
exp([out])	Calculate the exponential of all elements in the input array.
exp2([out])	Calculate 2**p for all p in the input array.
expm1([out])	Calculate exp(x) - 1 for all elements in the array.

Continued on next page

Table 2.236 – continued from previous page

floor([out])	Return the floor of the input, element-wise.
floor_divide(x2[, out])	Return the largest integer smaller or equal to the division of the inputs.
fmax(x2[, out])	Element-wise maximum of array elements.
fmin(x2[, out])	Element-wise minimum of array elements.
fmod(x2[, out])	Return the element-wise remainder of division.
greater(x2[, out])	Return the truth value of (x1 > x2) element-wise.
greater_equal(x2[, out])	Return the truth value of (x1 >= x2) element-wise.
hypot(x2[, out])	Given the "legs" of a right triangle, return its hypotenuse.
invert([out])	Compute bit-wise inversion, or bit-wise NOT, element-wise.
isfinite([out])	Test element-wise for finiteness (not infinity or not Not a Number).
isinf([out])	Test element-wise for positive or negative infinity.
isnan([out])	Test element-wise for NaN and return result as a boolean array.
left_shift(x2[, out])	Shift the bits of an integer to the left.
less(x2[, out])	Return the truth value of (x1 < x2) element-wise.
less_equal(x2[, out])	Return the truth value of (x1 =< x2) element-wise.
log([out])	Natural logarithm, element-wise.
log10([out])	Return the base 10 logarithm of the input array, element-wise.
log1p([out])	Return the natural logarithm of one plus the input array, element-wise.
log2([out])	Base-2 logarithm of x.
logaddexp(x2[, out])	Logarithm of the sum of exponentiations of the inputs.
logaddexp2(x2[, out])	Logarithm of the sum of exponentiations of the inputs in base-2.
logical_and(x2[, out])	Compute the truth value of x1 AND x2 element-wise.
logical_not([out])	Compute the truth value of NOT x element-wise.
logical_or(x2[, out])	Compute the truth value of x1 OR x2 element-wise.
logical_xor(x2[, out])	Compute the truth value of x1 XOR x2, element-wise.
max()	Maximum value in array.
maximum(x2[, out])	Element-wise maximum of array elements.
min()	Minimum value in array.
minimum(x2[, out])	Element-wise minimum of array elements.
mod(x2[, out])	Return element-wise remainder of division.
modf([out1, out2])	Return the fractional and integral parts of an array, element-wise.
multiply(x2[, out])	Multiply arguments element-wise with numpy.
negative([out])	Numerical negative, element-wise.
nextafter(x2[, out])	Return the next floating-point value after x1 towards x2, element-wise.
not_equal(x2[, out])	Return (x1 != x2) element-wise.
power(x2[, out])	First array elements raised to powers from second array, element-wise.
prod()	Product of array elements.
rad2deg([out])	Convert angles from radians to degrees.
reciprocal([out])	Return the reciprocal of the argument, element-wise.
remainder(x2[, out])	Return element-wise remainder of division.
right_shift(x2[, out])	Shift the bits of an integer to the right.
rint([out])	Round elements of the array to the nearest integer.
sign([out])	Returns an element-wise indication of the sign of a number.
signbit([out])	Returns element-wise True where signbit is set (less than zero).
sin([out])	Trigonometric sine, element-wise.
sinh([out])	Hyperbolic sine, element-wise.
sqrt([out])	Return the positive square-root of an array, element-wise.
square([out])	Return the element-wise square of the input.
subtract(x2[, out])	Subtract arguments, element-wise with numpy.
sum()	Sum of array elements.

Continued on next page

Table 2.236 – continued from previous page

tan([out])	Compute tangent element-wise.
tanh([out])	Compute hyperbolic tangent element-wise.
true_divide(x2[, out])	Returns a true division of the inputs, element-wise.
trunc([out])	Return the truncated value of the input, element-wise.

NtuplesBaseUFuncs.absolute

NtuplesBaseUFuncs.**absolute**(*out=None*)

Calculate the absolute value element-wise.

See also:

numpy.absolute

NtuplesBaseUFuncs.add

NtuplesBaseUFuncs.**add**(*x2*, *out=None*)

Add arguments element-wise with numpy.

See also:

numpy.add

NtuplesBaseUFuncs.arccos

NtuplesBaseUFuncs.**arccos**(*out=None*)

Trigonometric inverse cosine, element-wise.

See also:

numpy.arccos

NtuplesBaseUFuncs.arccosh

NtuplesBaseUFuncs.**arccosh**(*out=None*)

Inverse hyperbolic cosine, element-wise.

See also:

numpy.arccosh

NtuplesBaseUFuncs.arcsin

NtuplesBaseUFuncs.**arcsin**(*out=None*)

Inverse sine, element-wise.

See also:

numpy.arcsin

NtuplesBaseUFuncs.arcsinh

NtuplesBaseUFuncs.**arcsinh**(*out=None*)

Inverse hyperbolic sine element-wise.

See also:

numpy.arcsinh

NtuplesBaseUFuncs.arctan
NtuplesBaseUFuncs.**arctan**(*out=None*)
Trigonometric inverse tangent, element-wise.

See also:

numpy.arctan

NtuplesBaseUFuncs.arctan2
NtuplesBaseUFuncs.**arctan2**(*x2*, *out=None*)
Element-wise arc tangent of x1/x2 choosing the quadrant correctly.

See also:

numpy.arctan2

NtuplesBaseUFuncs.arctanh
NtuplesBaseUFuncs.**arctanh**(*out=None*)
Inverse hyperbolic tangent element-wise.

See also:

numpy.arctanh

NtuplesBaseUFuncs.bitwise_and
NtuplesBaseUFuncs.**bitwise_and**(*x2*, *out=None*)
Compute the bit-wise AND of two arrays element-wise.

See also:

numpy.bitwise_and

NtuplesBaseUFuncs.bitwise_or
NtuplesBaseUFuncs.**bitwise_or**(*x2*, *out=None*)
Compute the bit-wise OR of two arrays element-wise.

See also:

numpy.bitwise_or

NtuplesBaseUFuncs.bitwise_xor
NtuplesBaseUFuncs.**bitwise_xor**(*x2*, *out=None*)
Compute the bit-wise XOR of two arrays element-wise.

See also:

numpy.bitwise_xor

NtuplesBaseUFuncs.ceil
NtuplesBaseUFuncs.**ceil**(*out=None*)
Return the ceiling of the input, element-wise.

See also:

numpy.ceil

NtuplesBaseUFuncs.conj
NtuplesBaseUFuncs.**conj**(*out=None*)
Return the complex conjugate, element-wise.

See also:

numpy.conj

NtuplesBaseUFuncs.copysign
NtuplesBaseUFuncs.**copysign**(*x2*, *out=None*)
Change the sign of x1 to that of x2, element-wise.

See also:

numpy.copysign

NtuplesBaseUFuncs.cos
NtuplesBaseUFuncs.**cos**(*out=None*)
Cosine element-wise.

See also:

numpy.cos

NtuplesBaseUFuncs.cosh
NtuplesBaseUFuncs.**cosh**(*out=None*)
Hyperbolic cosine, element-wise.

See also:

numpy.cosh

NtuplesBaseUFuncs.deg2rad
NtuplesBaseUFuncs.**deg2rad**(*out=None*)
Convert angles from degrees to radians.

See also:

numpy.deg2rad

NtuplesBaseUFuncs.divide
NtuplesBaseUFuncs.**divide**(*x2*, *out=None*)
Divide arguments element-wise with numpy.

See also:

numpy.divide

NtuplesBaseUFuncs.equal
NtuplesBaseUFuncs.**equal**(*x2*, *out=None*)
Return (x1 == x2) element-wise.

See also:

numpy.equal

NtuplesBaseUFuncs.exp
NtuplesBaseUFuncs.**exp**(*out=None*)
Calculate the exponential of all elements in the input array.

See also:

numpy.exp

NtuplesBaseUFuncs.exp2
NtuplesBaseUFuncs.**exp2**(*out=None*)
Calculate 2**p for all p in the input array.

See also:

numpy.exp2

NtuplesBaseUFuncs.expm1
NtuplesBaseUFuncs.**expm1**(*out=None*)
Calculate exp(x) - 1 for all elements in the array.

See also:

numpy.expm1

NtuplesBaseUFuncs.floor
NtuplesBaseUFuncs.**floor**(*out=None*)
Return the floor of the input, element-wise.

See also:

numpy.floor

NtuplesBaseUFuncs.floor_divide
NtuplesBaseUFuncs.**floor_divide**(*x2*, *out=None*)
Return the largest integer smaller or equal to the division of the inputs.

See also:

numpy.floor_divide

NtuplesBaseUFuncs.fmax
NtuplesBaseUFuncs.**fmax**(*x2*, *out=None*)
Element-wise maximum of array elements.

See also:

numpy.fmax

NtuplesBaseUFuncs.fmin
NtuplesBaseUFuncs.**fmin**(*x2*, *out=None*)
Element-wise minimum of array elements.

See also:

numpy.fmin

NtuplesBaseUFuncs.fmod
NtuplesBaseUFuncs.**fmod**(*x2*, *out=None*)
Return the element-wise remainder of division.

See also:

numpy.fmod

NtuplesBaseUFuncs.greater
NtuplesBaseUFuncs.**greater**(*x2*, *out=None*)
Return the truth value of (x1 > x2) element-wise.

See also:

numpy.greater

NtuplesBaseUFuncs.greater_equal
NtuplesBaseUFuncs.**greater_equal**(*x2*, *out=None*)
Return the truth value of (x1 >= x2) element-wise.

See also:

numpy.greater_equal

NtuplesBaseUFuncs.hypot
NtuplesBaseUFuncs.**hypot**(*x2*, *out=None*)
Given the "legs" of a right triangle, return its hypotenuse.

See also:

numpy.hypot

NtuplesBaseUFuncs.invert
NtuplesBaseUFuncs.**invert**(*out=None*)
Compute bit-wise inversion, or bit-wise NOT, element-wise.

See also:

numpy.invert

NtuplesBaseUFuncs.isfinite
NtuplesBaseUFuncs.**isfinite**(*out=None*)
Test element-wise for finiteness (not infinity or not Not a Number).

See also:

numpy.isfinite

NtuplesBaseUFuncs.isinf
NtuplesBaseUFuncs.**isinf**(*out=None*)
Test element-wise for positive or negative infinity.

See also:

numpy.isinf

NtuplesBaseUFuncs.isnan
NtuplesBaseUFuncs.**isnan**(*out=None*)
Test element-wise for NaN and return result as a boolean array.

See also:

numpy.isnan

NtuplesBaseUFuncs.left_shift
NtuplesBaseUFuncs.**left_shift**(*x2*, *out=None*)
Shift the bits of an integer to the left.

See also:

numpy.left_shift

NtuplesBaseUFuncs.less
NtuplesBaseUFuncs.**less**(*x2*, *out=None*)
Return the truth value of (x1 < x2) element-wise.

See also:

numpy.less

NtuplesBaseUFuncs.less_equal
NtuplesBaseUFuncs.**less_equal**(*x2*, *out=None*)
Return the truth value of (x1 =< x2) element-wise.

See also:

numpy.less_equal

NtuplesBaseUFuncs.log
NtuplesBaseUFuncs.**log**(*out=None*)
Natural logarithm, element-wise.

See also:

numpy.log

NtuplesBaseUFuncs.log10
NtuplesBaseUFuncs.**log10**(*out=None*)
Return the base 10 logarithm of the input array, element-wise.

See also:

numpy.log10

NtuplesBaseUFuncs.log1p
NtuplesBaseUFuncs.**log1p**(*out=None*)
Return the natural logarithm of one plus the input array, element-wise.

See also:

numpy.log1p

NtuplesBaseUFuncs.log2
NtuplesBaseUFuncs.**log2**(*out=None*)
Base-2 logarithm of x.

See also:

numpy.log2

NtuplesBaseUFuncs.logaddexp
NtuplesBaseUFuncs.**logaddexp**(*x2*, *out=None*)
Logarithm of the sum of exponentiations of the inputs.

See also:

numpy.logaddexp

NtuplesBaseUFuncs.logaddexp2
NtuplesBaseUFuncs.**logaddexp2**(*x2*, *out=None*)
Logarithm of the sum of exponentiations of the inputs in base-2.

See also:

numpy.logaddexp2

NtuplesBaseUFuncs.logical_and
NtuplesBaseUFuncs.**logical_and**(*x2*, *out=None*)
Compute the truth value of x1 AND x2 element-wise.

See also:

numpy.logical_and

NtuplesBaseUFuncs.logical_not
NtuplesBaseUFuncs.**logical_not**(*out=None*)
Compute the truth value of NOT x element-wise.

See also:

numpy.logical_not

NtuplesBaseUFuncs.logical_or
NtuplesBaseUFuncs.**logical_or**(*x2*, *out=None*)
Compute the truth value of x1 OR x2 element-wise.

See also:

numpy.logical_or

NtuplesBaseUFuncs.logical_xor
NtuplesBaseUFuncs.**logical_xor**(*x2*, *out=None*)
Compute the truth value of x1 XOR x2, element-wise.

See also:

numpy.logical_xor

NtuplesBaseUFuncs.max
NtuplesBaseUFuncs.**max**()
Maximum value in array.

See also:

numpy.max

NtuplesBaseUFuncs.maximum
NtuplesBaseUFuncs.**maximum**(*x2*, *out=None*)
Element-wise maximum of array elements.

See also:

numpy.maximum

NtuplesBaseUFuncs.min
NtuplesBaseUFuncs.**min**()
Minimum value in array.

See also:

numpy.min

NtuplesBaseUFuncs.minimum
NtuplesBaseUFuncs.**minimum**(*x2*, *out=None*)
Element-wise minimum of array elements.

See also:

numpy.minimum

NtuplesBaseUFuncs.mod
NtuplesBaseUFuncs.**mod**(*x2*, *out=None*)
Return element-wise remainder of division.

See also:

numpy.mod

NtuplesBaseUFuncs.modf
NtuplesBaseUFuncs.**modf**(*out1=None*, *out2=None*)
Return the fractional and integral parts of an array, element-wise.

See also:

numpy.modf

NtuplesBaseUFuncs.multiply
NtuplesBaseUFuncs.**multiply**(*x2*, *out=None*)
Multiply arguments element-wise with numpy.

See also:

numpy.multiply

NtuplesBaseUFuncs.negative
NtuplesBaseUFuncs.**negative**(*out=None*)
Numerical negative, element-wise.

See also:

numpy.negative

NtuplesBaseUFuncs.nextafter
NtuplesBaseUFuncs.**nextafter**(*x2*, *out=None*)
Return the next floating-point value after x1 towards x2, element-wise.

See also:

numpy.nextafter

NtuplesBaseUFuncs.not_equal
NtuplesBaseUFuncs.**not_equal**(*x2*, *out=None*)
Return (x1 != x2) element-wise.

See also:

numpy.not_equal

NtuplesBaseUFuncs.power
NtuplesBaseUFuncs.**power**(*x2*, *out=None*)
First array elements raised to powers from second array, element-wise.

See also:

numpy.power

NtuplesBaseUFuncs.prod
NtuplesBaseUFuncs.**prod**()
Product of array elements.

See also:

numpy.prod

NtuplesBaseUFuncs.rad2deg
NtuplesBaseUFuncs.**rad2deg**(*out=None*)
Convert angles from radians to degrees.

See also:

numpy.rad2deg

NtuplesBaseUFuncs.reciprocal
NtuplesBaseUFuncs.**reciprocal**(*out=None*)
Return the reciprocal of the argument, element-wise.

See also:

numpy.reciprocal

NtuplesBaseUFuncs.remainder
NtuplesBaseUFuncs.**remainder**(*x2*, *out=None*)
Return element-wise remainder of division.

See also:

numpy.remainder

NtuplesBaseUFuncs.right_shift
NtuplesBaseUFuncs.**right_shift**(*x2*, *out=None*)
Shift the bits of an integer to the right.

See also:

numpy.right_shift

NtuplesBaseUFuncs.rint
NtuplesBaseUFuncs.**rint**(*out=None*)
Round elements of the array to the nearest integer.

See also:

numpy.rint

NtuplesBaseUFuncs.sign
NtuplesBaseUFuncs.**sign**(*out=None*)
Returns an element-wise indication of the sign of a number.

See also:

numpy.sign

NtuplesBaseUFuncs.signbit
NtuplesBaseUFuncs.**signbit**(*out=None*)
Returns element-wise True where signbit is set (less than zero).

See also:

numpy.signbit

NtuplesBaseUFuncs.sin
NtuplesBaseUFuncs.**sin**(*out=None*)
Trigonometric sine, element-wise.

See also:

numpy.sin

NtuplesBaseUFuncs.sinh
NtuplesBaseUFuncs.**sinh**(*out=None*)
Hyperbolic sine, element-wise.

See also:

numpy.sinh

NtuplesBaseUFuncs.sqrt
NtuplesBaseUFuncs.**sqrt**(*out=None*)
Return the positive square-root of an array, element-wise.

See also:

numpy.sqrt

NtuplesBaseUFuncs.square
NtuplesBaseUFuncs.**square**(*out=None*)
Return the element-wise square of the input.

See also:

numpy.square

NtuplesBaseUFuncs.subtract
NtuplesBaseUFuncs.**subtract**(*x2*, *out=None*)
Subtract arguments, element-wise with numpy.

See also:

numpy.subtract

NtuplesBaseUFuncs.sum
NtuplesBaseUFuncs.**sum**()
Sum of array elements.

See also:

numpy.sum

NtuplesBaseUFuncs.tan
NtuplesBaseUFuncs.**tan**(*out=None*)
Compute tangent element-wise.

See also:

numpy.tan

NtuplesBaseUFuncs.tanh
NtuplesBaseUFuncs.**tanh**(*out=None*)
Compute hyperbolic tangent element-wise.

See also:

numpy.tanh

NtuplesBaseUFuncs.true_divide
NtuplesBaseUFuncs.**true_divide**(*x2*, *out=None*)
Returns a true division of the inputs, element-wise.

See also:

numpy.true_divide

NtuplesBaseUFuncs.trunc
NtuplesBaseUFuncs.**trunc**(*out=None*)
Return the truncated value of the input, element-wise.

See also:

numpy.trunc

__init__(*vector*)
Create ufunc wrapper for vector.

NtuplesUFuncs
class odl.util.ufuncs.**NtuplesUFuncs**(*vector*)
Bases: *odl.util.ufuncs.NtuplesBaseUFuncs*

UFuncs for *NtuplesVector* objects.

Internal object, should not be created except in *NtuplesVector*.

Methods

__eq__	Return self==value.
absolute([out])	Calculate the absolute value element-wise.
add(x2[, out])	Add arguments element-wise with numpy.
arccos([out])	Trigonometric inverse cosine, element-wise.
arccosh([out])	Inverse hyperbolic cosine, element-wise.
arcsin([out])	Inverse sine, element-wise.
arcsinh([out])	Inverse hyperbolic sine element-wise.
arctan([out])	Trigonometric inverse tangent, element-wise.
arctan2(x2[, out])	Element-wise arc tangent of x1/x2 choosing the quadrant correctly.
arctanh([out])	Inverse hyperbolic tangent element-wise.
bitwise_and(x2[, out])	Compute the bit-wise AND of two arrays element-wise.
bitwise_or(x2[, out])	Compute the bit-wise OR of two arrays element-wise.
bitwise_xor(x2[, out])	Compute the bit-wise XOR of two arrays element-wise.
ceil([out])	Return the ceiling of the input, element-wise.
conj([out])	Return the complex conjugate, element-wise.
copysign(x2[, out])	Change the sign of x1 to that of x2, element-wise.
cos([out])	Cosine element-wise.
cosh([out])	Hyperbolic cosine, element-wise.
deg2rad([out])	Convert angles from degrees to radians.
divide(x2[, out])	Divide arguments element-wise with numpy.
equal(x2[, out])	Return (x1 == x2) element-wise.
exp([out])	Calculate the exponential of all elements in the input array.
exp2([out])	Calculate 2**p for all p in the input array.
expm1([out])	Calculate exp(x) - 1 for all elements in the array.
floor([out])	Return the floor of the input, element-wise.
floor_divide(x2[, out])	Return the largest integer smaller or equal to the division of the inputs.
fmax(x2[, out])	Element-wise maximum of array elements.
fmin(x2[, out])	Element-wise minimum of array elements.
fmod(x2[, out])	Return the element-wise remainder of division.
greater(x2[, out])	Return the truth value of (x1 > x2) element-wise.
greater_equal(x2[, out])	Return the truth value of (x1 >= x2) element-wise.
hypot(x2[, out])	Given the "legs" of a right triangle, return its hypotenuse.

Continued on next page

Table 2.237 – continued from previous page

invert([out])	Compute bit-wise inversion, or bit-wise NOT, element-wise.
isfinite([out])	Test element-wise for finiteness (not infinity or not Not a Number).
isinf([out])	Test element-wise for positive or negative infinity.
isnan([out])	Test element-wise for NaN and return result as a boolean array.
left_shift(x2[, out])	Shift the bits of an integer to the left.
less(x2[, out])	Return the truth value of (x1 < x2) element-wise.
less_equal(x2[, out])	Return the truth value of (x1 =< x2) element-wise.
log([out])	Natural logarithm, element-wise.
log10([out])	Return the base 10 logarithm of the input array, element-wise.
log1p([out])	Return the natural logarithm of one plus the input array, element-wise.
log2([out])	Base-2 logarithm of x.
logaddexp(x2[, out])	Logarithm of the sum of exponentiations of the inputs.
logaddexp2(x2[, out])	Logarithm of the sum of exponentiations of the inputs in base-2.
logical_and(x2[, out])	Compute the truth value of x1 AND x2 element-wise.
logical_not([out])	Compute the truth value of NOT x element-wise.
logical_or(x2[, out])	Compute the truth value of x1 OR x2 element-wise.
logical_xor(x2[, out])	Compute the truth value of x1 XOR x2, element-wise.
max()	Maximum value in array.
maximum(x2[, out])	Element-wise maximum of array elements.
min()	Minimum value in array.
minimum(x2[, out])	Element-wise minimum of array elements.
mod(x2[, out])	Return element-wise remainder of division.
modf([out1, out2])	Return the fractional and integral parts of an array, element-wise.
multiply(x2[, out])	Multiply arguments element-wise with numpy.
negative([out])	Numerical negative, element-wise.
nextafter(x2[, out])	Return the next floating-point value after x1 towards x2, element-wise.
not_equal(x2[, out])	Return (x1 != x2) element-wise.
power(x2[, out])	First array elements raised to powers from second array, element-wise.
prod()	Product of array elements.
rad2deg([out])	Convert angles from radians to degrees.
reciprocal([out])	Return the reciprocal of the argument, element-wise.
remainder(x2[, out])	Return element-wise remainder of division.
right_shift(x2[, out])	Shift the bits of an integer to the right.
rint([out])	Round elements of the array to the nearest integer.
sign([out])	Returns an element-wise indication of the sign of a number.
signbit([out])	Returns element-wise True where signbit is set (less than zero).
sin([out])	Trigonometric sine, element-wise.
sinh([out])	Hyperbolic sine, element-wise.
sqrt([out])	Return the positive square-root of an array, element-wise.
square([out])	Return the element-wise square of the input.
subtract(x2[, out])	Subtract arguments, element-wise with numpy.
sum()	Sum of array elements.
tan([out])	Compute tangent element-wise.
tanh([out])	Compute hyperbolic tangent element-wise.
true_divide(x2[, out])	Returns a true division of the inputs, element-wise.
trunc([out])	Return the truncated value of the input, element-wise.

NtuplesUFuncs.absolute

NtuplesUFuncs.**absolute**(*out=None*)

Calculate the absolute value element-wise.

See also:

`numpy.absolute`

NtuplesUFuncs.add

`NtuplesUFuncs.`**`add`**(*x2*, *out=None*)

Add arguments element-wise with numpy.

See also:

`numpy.add`

NtuplesUFuncs.arccos

`NtuplesUFuncs.`**`arccos`**(*out=None*)

Trigonometric inverse cosine, element-wise.

See also:

`numpy.arccos`

NtuplesUFuncs.arccosh

`NtuplesUFuncs.`**`arccosh`**(*out=None*)

Inverse hyperbolic cosine, element-wise.

See also:

`numpy.arccosh`

NtuplesUFuncs.arcsin

`NtuplesUFuncs.`**`arcsin`**(*out=None*)

Inverse sine, element-wise.

See also:

`numpy.arcsin`

NtuplesUFuncs.arcsinh

`NtuplesUFuncs.`**`arcsinh`**(*out=None*)

Inverse hyperbolic sine element-wise.

See also:

`numpy.arcsinh`

NtuplesUFuncs.arctan

`NtuplesUFuncs.`**`arctan`**(*out=None*)

Trigonometric inverse tangent, element-wise.

See also:

`numpy.arctan`

NtuplesUFuncs.arctan2
NtuplesUFuncs.**arctan2**(*x2*, *out=None*)
Element-wise arc tangent of x1/x2 choosing the quadrant correctly.

See also:

numpy.arctan2

NtuplesUFuncs.arctanh
NtuplesUFuncs.**arctanh**(*out=None*)
Inverse hyperbolic tangent element-wise.

See also:

numpy.arctanh

NtuplesUFuncs.bitwise_and
NtuplesUFuncs.**bitwise_and**(*x2*, *out=None*)
Compute the bit-wise AND of two arrays element-wise.

See also:

numpy.bitwise_and

NtuplesUFuncs.bitwise_or
NtuplesUFuncs.**bitwise_or**(*x2*, *out=None*)
Compute the bit-wise OR of two arrays element-wise.

See also:

numpy.bitwise_or

NtuplesUFuncs.bitwise_xor
NtuplesUFuncs.**bitwise_xor**(*x2*, *out=None*)
Compute the bit-wise XOR of two arrays element-wise.

See also:

numpy.bitwise_xor

NtuplesUFuncs.ceil
NtuplesUFuncs.**ceil**(*out=None*)
Return the ceiling of the input, element-wise.

See also:

numpy.ceil

NtuplesUFuncs.conj
NtuplesUFuncs.**conj**(*out=None*)
Return the complex conjugate, element-wise.

See also:

numpy.conj

NtuplesUFuncs.copysign
NtuplesUFuncs.**copysign**(*x2*, *out=None*)
Change the sign of x1 to that of x2, element-wise.

See also:

numpy.copysign

NtuplesUFuncs.cos
NtuplesUFuncs.**cos**(*out=None*)
Cosine element-wise.

See also:

numpy.cos

NtuplesUFuncs.cosh
NtuplesUFuncs.**cosh**(*out=None*)
Hyperbolic cosine, element-wise.

See also:

numpy.cosh

NtuplesUFuncs.deg2rad
NtuplesUFuncs.**deg2rad**(*out=None*)
Convert angles from degrees to radians.

See also:

numpy.deg2rad

NtuplesUFuncs.divide
NtuplesUFuncs.**divide**(*x2*, *out=None*)
Divide arguments element-wise with numpy.

See also:

numpy.divide

NtuplesUFuncs.equal
NtuplesUFuncs.**equal**(*x2*, *out=None*)
Return (x1 == x2) element-wise.

See also:

numpy.equal

NtuplesUFuncs.exp
NtuplesUFuncs.**exp**(*out=None*)
Calculate the exponential of all elements in the input array.

See also:

numpy.exp

NtuplesUFuncs.exp2
NtuplesUFuncs.**exp2**(*out=None*)
Calculate 2**p for all p in the input array.

See also:

numpy.exp2

NtuplesUFuncs.expm1
NtuplesUFuncs.**expm1**(*out=None*)
Calculate exp(x) - 1 for all elements in the array.

See also:

numpy.expm1

NtuplesUFuncs.floor
NtuplesUFuncs.**floor**(*out=None*)
Return the floor of the input, element-wise.

See also:

numpy.floor

NtuplesUFuncs.floor_divide
NtuplesUFuncs.**floor_divide**(*x2*, *out=None*)
Return the largest integer smaller or equal to the division of the inputs.

See also:

numpy.floor_divide

NtuplesUFuncs.fmax
NtuplesUFuncs.**fmax**(*x2*, *out=None*)
Element-wise maximum of array elements.

See also:

numpy.fmax

NtuplesUFuncs.fmin
NtuplesUFuncs.**fmin**(*x2*, *out=None*)
Element-wise minimum of array elements.

See also:

numpy.fmin

NtuplesUFuncs.fmod
NtuplesUFuncs.**fmod**(*x2*, *out=None*)
Return the element-wise remainder of division.

See also:

numpy.fmod

NtuplesUFuncs.greater
NtuplesUFuncs.**greater**(*x2*, *out=None*)
Return the truth value of (x1 > x2) element-wise.

See also:

numpy.greater

NtuplesUFuncs.greater_equal
NtuplesUFuncs.**greater_equal**(*x2*, *out=None*)
Return the truth value of (x1 >= x2) element-wise.

See also:

numpy.greater_equal

NtuplesUFuncs.hypot
NtuplesUFuncs.**hypot**(*x2*, *out=None*)
Given the "legs" of a right triangle, return its hypotenuse.

See also:

numpy.hypot

NtuplesUFuncs.invert
NtuplesUFuncs.**invert**(*out=None*)
Compute bit-wise inversion, or bit-wise NOT, element-wise.

See also:

numpy.invert

NtuplesUFuncs.isfinite
NtuplesUFuncs.**isfinite**(*out=None*)
Test element-wise for finiteness (not infinity or not Not a Number).

See also:

numpy.isfinite

NtuplesUFuncs.isinf
NtuplesUFuncs.**isinf**(*out=None*)
Test element-wise for positive or negative infinity.

See also:

numpy.isinf

NtuplesUFuncs.isnan
NtuplesUFuncs.**isnan**(*out=None*)
Test element-wise for NaN and return result as a boolean array.

See also:

numpy.isnan

NtuplesUFuncs.left_shift
NtuplesUFuncs.**left_shift**(*x2*, *out=None*)
Shift the bits of an integer to the left.

See also:

numpy.left_shift

NtuplesUFuncs.less
NtuplesUFuncs.**less**(*x2*, *out=None*)
Return the truth value of (x1 < x2) element-wise.

See also:

numpy.less

NtuplesUFuncs.less_equal
NtuplesUFuncs.**less_equal**(*x2*, *out=None*)
Return the truth value of (x1 =< x2) element-wise.

See also:

numpy.less_equal

NtuplesUFuncs.log
NtuplesUFuncs.**log**(*out=None*)
Natural logarithm, element-wise.

See also:

numpy.log

NtuplesUFuncs.log10
NtuplesUFuncs.**log10**(*out=None*)
Return the base 10 logarithm of the input array, element-wise.

See also:

numpy.log10

NtuplesUFuncs.log1p
NtuplesUFuncs.**log1p**(*out=None*)
Return the natural logarithm of one plus the input array, element-wise.

See also:

numpy.log1p

NtuplesUFuncs.log2
NtuplesUFuncs.**log2**(*out=None*)
Base-2 logarithm of x.

See also:

numpy.log2

NtuplesUFuncs.logaddexp
NtuplesUFuncs.**logaddexp**(*x2*, *out=None*)
Logarithm of the sum of exponentiations of the inputs.

See also:

numpy.logaddexp

NtuplesUFuncs.logaddexp2
NtuplesUFuncs.**logaddexp2**(*x2*, *out=None*)
Logarithm of the sum of exponentiations of the inputs in base-2.

See also:

numpy.logaddexp2

NtuplesUFuncs.logical_and
NtuplesUFuncs.**logical_and**(*x2*, *out=None*)
Compute the truth value of x1 AND x2 element-wise.

See also:

numpy.logical_and

NtuplesUFuncs.logical_not
NtuplesUFuncs.**logical_not**(*out=None*)
Compute the truth value of NOT x element-wise.

See also:

numpy.logical_not

NtuplesUFuncs.logical_or
NtuplesUFuncs.**logical_or**(*x2*, *out=None*)
Compute the truth value of x1 OR x2 element-wise.

See also:

numpy.logical_or

NtuplesUFuncs.logical_xor
NtuplesUFuncs.**logical_xor**(*x2*, *out=None*)
Compute the truth value of x1 XOR x2, element-wise.

See also:

numpy.logical_xor

NtuplesUFuncs.max
NtuplesUFuncs.**max**()
Maximum value in array.

See also:

numpy.max

NtuplesUFuncs.maximum
NtuplesUFuncs.**maximum**(*x2*, *out=None*)
Element-wise maximum of array elements.

See also:

numpy.maximum

NtuplesUFuncs.min
NtuplesUFuncs.**min**()
Minimum value in array.

See also:

numpy.min

NtuplesUFuncs.minimum
NtuplesUFuncs.**minimum**(*x2*, *out=None*)
Element-wise minimum of array elements.

See also:

numpy.minimum

NtuplesUFuncs.mod
NtuplesUFuncs.**mod**(*x2*, *out=None*)
Return element-wise remainder of division.

See also:

numpy.mod

NtuplesUFuncs.modf
NtuplesUFuncs.**modf**(*out1=None*, *out2=None*)
Return the fractional and integral parts of an array, element-wise.

See also:

numpy.modf

NtuplesUFuncs.multiply
NtuplesUFuncs.**multiply**(*x2*, *out=None*)
Multiply arguments element-wise with numpy.

See also:

numpy.multiply

NtuplesUFuncs.negative
NtuplesUFuncs.**negative**(*out=None*)
Numerical negative, element-wise.

See also:

numpy.negative

NtuplesUFuncs.nextafter
NtuplesUFuncs.**nextafter**(*x2*, *out=None*)
Return the next floating-point value after x1 towards x2, element-wise.

See also:

numpy.nextafter

NtuplesUFuncs.not_equal
NtuplesUFuncs.**not_equal**(*x2*, *out=None*)
Return (x1 != x2) element-wise.

See also:

numpy.not_equal

NtuplesUFuncs.power
NtuplesUFuncs.**power**(*x2*, *out=None*)
First array elements raised to powers from second array, element-wise.

See also:

numpy.power

NtuplesUFuncs.prod
NtuplesUFuncs.**prod**()
Product of array elements.

See also:

numpy.prod

NtuplesUFuncs.rad2deg
NtuplesUFuncs.**rad2deg**(*out=None*)
Convert angles from radians to degrees.

See also:

numpy.rad2deg

NtuplesUFuncs.reciprocal
NtuplesUFuncs.**reciprocal**(*out=None*)
Return the reciprocal of the argument, element-wise.

See also:

numpy.reciprocal

NtuplesUFuncs.remainder
NtuplesUFuncs.**remainder**(*x2*, *out=None*)
Return element-wise remainder of division.

See also:

numpy.remainder

NtuplesUFuncs.right_shift
NtuplesUFuncs.**right_shift**(*x2*, *out=None*)
Shift the bits of an integer to the right.

See also:

numpy.right_shift

NtuplesUFuncs.rint
NtuplesUFuncs.**rint**(*out=None*)
Round elements of the array to the nearest integer.

See also:

numpy.rint

NtuplesUFuncs.sign
NtuplesUFuncs.**sign**(*out=None*)
Returns an element-wise indication of the sign of a number.

See also:

numpy.sign

NtuplesUFuncs.signbit
NtuplesUFuncs.**signbit**(*out=None*)
Returns element-wise True where signbit is set (less than zero).

See also:

numpy.signbit

NtuplesUFuncs.sin
NtuplesUFuncs.**sin**(*out=None*)
Trigonometric sine, element-wise.

See also:

numpy.sin

NtuplesUFuncs.sinh
NtuplesUFuncs.**sinh**(*out=None*)
Hyperbolic sine, element-wise.

See also:

numpy.sinh

NtuplesUFuncs.sqrt
NtuplesUFuncs.**sqrt**(*out=None*)
Return the positive square-root of an array, element-wise.

See also:

numpy.sqrt

NtuplesUFuncs.square
NtuplesUFuncs.**square**(*out=None*)
Return the element-wise square of the input.

See also:

numpy.square

NtuplesUFuncs.subtract
NtuplesUFuncs.**subtract**(*x2*, *out=None*)
Subtract arguments, element-wise with numpy.

See also:

numpy.subtract

NtuplesUFuncs.sum
NtuplesUFuncs.**sum**()
Sum of array elements.

See also:

numpy.sum

NtuplesUFuncs.tan
NtuplesUFuncs.**tan**(*out=None*)
Compute tangent element-wise.

See also:

numpy.tan

NtuplesUFuncs.tanh
NtuplesUFuncs.**tanh**(*out=None*)
Compute hyperbolic tangent element-wise.

See also:

numpy.tanh

NtuplesUFuncs.true_divide
NtuplesUFuncs.**true_divide**(*x2*, *out=None*)
Returns a true division of the inputs, element-wise.

See also:

numpy.true_divide

NtuplesUFuncs.trunc
NtuplesUFuncs.**trunc**(*out=None*)
Return the truncated value of the input, element-wise.

See also:

numpy.trunc
__init__(*vector*)
Create ufunc wrapper for vector.

ProductSpaceUFuncs

class odl.util.ufuncs.**ProductSpaceUFuncs**(*vector*)

Bases: object

UFuncs for *ProductSpaceVector* objects.

Internal object, should not be created except in *ProductSpaceVector*.

Methods

__eq__	Return self==value.
absolute([out])	Calculate the absolute value element-wise.
add(x2[, out])	Add arguments element-wise with numpy.
arccos([out])	Trigonometric inverse cosine, element-wise.
arccosh([out])	Inverse hyperbolic cosine, element-wise.
arcsin([out])	Inverse sine, element-wise.
arcsinh([out])	Inverse hyperbolic sine element-wise.
arctan([out])	Trigonometric inverse tangent, element-wise.
arctan2(x2[, out])	Element-wise arc tangent of x1/x2 choosing the quadrant correctly.
arctanh([out])	Inverse hyperbolic tangent element-wise.
bitwise_and(x2[, out])	Compute the bit-wise AND of two arrays element-wise.
bitwise_or(x2[, out])	Compute the bit-wise OR of two arrays element-wise.
bitwise_xor(x2[, out])	Compute the bit-wise XOR of two arrays element-wise.
ceil([out])	Return the ceiling of the input, element-wise.
conj([out])	Return the complex conjugate, element-wise.
copysign(x2[, out])	Change the sign of x1 to that of x2, element-wise.
cos([out])	Cosine element-wise.
cosh([out])	Hyperbolic cosine, element-wise.
deg2rad([out])	Convert angles from degrees to radians.
divide(x2[, out])	Divide arguments element-wise with numpy.
equal(x2[, out])	Return (x1 == x2) element-wise.
exp([out])	Calculate the exponential of all elements in the input array.
exp2([out])	Calculate 2**p for all p in the input array.
expm1([out])	Calculate exp(x) - 1 for all elements in the array.
floor([out])	Return the floor of the input, element-wise.
floor_divide(x2[, out])	Return the largest integer smaller or equal to the division of the inputs.
fmax(x2[, out])	Element-wise maximum of array elements.
fmin(x2[, out])	Element-wise minimum of array elements.
fmod(x2[, out])	Return the element-wise remainder of division.
greater(x2[, out])	Return the truth value of (x1 > x2) element-wise.
greater_equal(x2[, out])	Return the truth value of (x1 >= x2) element-wise.
hypot(x2[, out])	Given the "legs" of a right triangle, return its hypotenuse.
invert([out])	Compute bit-wise inversion, or bit-wise NOT, element-wise.
isfinite([out])	Test element-wise for finiteness (not infinity or not Not a Number).
isinf([out])	Test element-wise for positive or negative infinity.
isnan([out])	Test element-wise for NaN and return result as a boolean array.
left_shift(x2[, out])	Shift the bits of an integer to the left.
less(x2[, out])	Return the truth value of (x1 < x2) element-wise.
less_equal(x2[, out])	Return the truth value of (x1 =< x2) element-wise.
log([out])	Natural logarithm, element-wise.
log10([out])	Return the base 10 logarithm of the input array, element-wise.

Continued on next page

Table 2.238 – continued from previous page

log1p([out])	Return the natural logarithm of one plus the input array, element-wise.
log2([out])	Base-2 logarithm of x.
logaddexp(x2[, out])	Logarithm of the sum of exponentiations of the inputs.
logaddexp2(x2[, out])	Logarithm of the sum of exponentiations of the inputs in base-2.
logical_and(x2[, out])	Compute the truth value of x1 AND x2 element-wise.
logical_not([out])	Compute the truth value of NOT x element-wise.
logical_or(x2[, out])	Compute the truth value of x1 OR x2 element-wise.
logical_xor(x2[, out])	Compute the truth value of x1 XOR x2, element-wise.
max()	Maximum value in array.
maximum(x2[, out])	Element-wise maximum of array elements.
min()	Minimum value in array.
minimum(x2[, out])	Element-wise minimum of array elements.
mod(x2[, out])	Return element-wise remainder of division.
modf([out1, out2])	Return the fractional and integral parts of an array, element-wise.
multiply(x2[, out])	Multiply arguments element-wise with numpy.
negative([out])	Numerical negative, element-wise.
nextafter(x2[, out])	Return the next floating-point value after x1 towards x2, element-wise.
not_equal(x2[, out])	Return (x1 != x2) element-wise.
power(x2[, out])	First array elements raised to powers from second array, element-wise.
prod()	Product of array elements.
rad2deg([out])	Convert angles from radians to degrees.
reciprocal([out])	Return the reciprocal of the argument, element-wise.
remainder(x2[, out])	Return element-wise remainder of division.
right_shift(x2[, out])	Shift the bits of an integer to the right.
rint([out])	Round elements of the array to the nearest integer.
sign([out])	Returns an element-wise indication of the sign of a number.
signbit([out])	Returns element-wise True where signbit is set (less than zero).
sin([out])	Trigonometric sine, element-wise.
sinh([out])	Hyperbolic sine, element-wise.
sqrt([out])	Return the positive square-root of an array, element-wise.
square([out])	Return the element-wise square of the input.
subtract(x2[, out])	Subtract arguments, element-wise with numpy.
sum()	Sum of array elements.
tan([out])	Compute tangent element-wise.
tanh([out])	Compute hyperbolic tangent element-wise.
true_divide(x2[, out])	Returns a true division of the inputs, element-wise.
trunc([out])	Return the truncated value of the input, element-wise.

ProductSpaceUFuncs.absolute

`ProductSpaceUFuncs.`**`absolute`**(*out=None*)

Calculate the absolute value element-wise.

See also:

`numpy.absolute`

ProductSpaceUFuncs.add

`ProductSpaceUFuncs.`**`add`**(*x2*, *out=None*)

Add arguments element-wise with numpy.

See also:

`numpy.add`

ProductSpaceUFuncs.arccos
`ProductSpaceUFuncs.`**`arccos`**(*out=None*)
Trigonometric inverse cosine, element-wise.

See also:

`numpy.arccos`

ProductSpaceUFuncs.arccosh
`ProductSpaceUFuncs.`**`arccosh`**(*out=None*)
Inverse hyperbolic cosine, element-wise.

See also:

`numpy.arccosh`

ProductSpaceUFuncs.arcsin
`ProductSpaceUFuncs.`**`arcsin`**(*out=None*)
Inverse sine, element-wise.

See also:

`numpy.arcsin`

ProductSpaceUFuncs.arcsinh
`ProductSpaceUFuncs.`**`arcsinh`**(*out=None*)
Inverse hyperbolic sine element-wise.

See also:

`numpy.arcsinh`

ProductSpaceUFuncs.arctan
`ProductSpaceUFuncs.`**`arctan`**(*out=None*)
Trigonometric inverse tangent, element-wise.

See also:

`numpy.arctan`

ProductSpaceUFuncs.arctan2
`ProductSpaceUFuncs.`**`arctan2`**(*x2*, *out=None*)
Element-wise arc tangent of x1/x2 choosing the quadrant correctly.

See also:

`numpy.arctan2`

ProductSpaceUFuncs.arctanh
`ProductSpaceUFuncs.`**`arctanh`**(*out=None*)
Inverse hyperbolic tangent element-wise.

See also:

`numpy.arctanh`

ProductSpaceUFuncs.bitwise_and
`ProductSpaceUFuncs.`**`bitwise_and`**(*x2*, *out=None*)
Compute the bit-wise AND of two arrays element-wise.

See also:

`numpy.bitwise_and`

ProductSpaceUFuncs.bitwise_or
`ProductSpaceUFuncs.`**`bitwise_or`**(*x2*, *out=None*)
Compute the bit-wise OR of two arrays element-wise.

See also:

`numpy.bitwise_or`

ProductSpaceUFuncs.bitwise_xor
`ProductSpaceUFuncs.`**`bitwise_xor`**(*x2*, *out=None*)
Compute the bit-wise XOR of two arrays element-wise.

See also:

`numpy.bitwise_xor`

ProductSpaceUFuncs.ceil
`ProductSpaceUFuncs.`**`ceil`**(*out=None*)
Return the ceiling of the input, element-wise.

See also:

`numpy.ceil`

ProductSpaceUFuncs.conj
`ProductSpaceUFuncs.`**`conj`**(*out=None*)
Return the complex conjugate, element-wise.

See also:

`numpy.conj`

ProductSpaceUFuncs.copysign
`ProductSpaceUFuncs.`**`copysign`**(*x2*, *out=None*)
Change the sign of x1 to that of x2, element-wise.

See also:

`numpy.copysign`

ProductSpaceUFuncs.cos
`ProductSpaceUFuncs.`**`cos`**(*out=None*)
Cosine element-wise.

See also:

`numpy.cos`

ProductSpaceUFuncs.cosh
`ProductSpaceUFuncs.`**`cosh`**(*out=None*)
Hyperbolic cosine, element-wise.

See also:

`numpy.cosh`

ProductSpaceUFuncs.deg2rad
`ProductSpaceUFuncs.`**`deg2rad`**(*out=None*)
Convert angles from degrees to radians.

See also:

`numpy.deg2rad`

ProductSpaceUFuncs.divide
`ProductSpaceUFuncs.`**`divide`**(*x2*, *out=None*)
Divide arguments element-wise with numpy.

See also:

`numpy.divide`

ProductSpaceUFuncs.equal
`ProductSpaceUFuncs.`**`equal`**(*x2*, *out=None*)
Return (x1 == x2) element-wise.

See also:

`numpy.equal`

ProductSpaceUFuncs.exp
`ProductSpaceUFuncs.`**`exp`**(*out=None*)
Calculate the exponential of all elements in the input array.

See also:

`numpy.exp`

ProductSpaceUFuncs.exp2
`ProductSpaceUFuncs.`**`exp2`**(*out=None*)
Calculate 2**p for all p in the input array.

See also:

`numpy.exp2`

ProductSpaceUFuncs.expm1
`ProductSpaceUFuncs.`**`expm1`**(*out=None*)
Calculate exp(x) - 1 for all elements in the array.

See also:

`numpy.expm1`

ProductSpaceUFuncs.floor
`ProductSpaceUFuncs.`**`floor`**(*out=None*)
Return the floor of the input, element-wise.

See also:

`numpy.floor`

ProductSpaceUFuncs.floor_divide
`ProductSpaceUFuncs.`**`floor_divide`**(*x2*, *out=None*)
Return the largest integer smaller or equal to the division of the inputs.

See also:

`numpy.floor_divide`

ProductSpaceUFuncs.fmax
`ProductSpaceUFuncs.`**`fmax`**(*x2*, *out=None*)
Element-wise maximum of array elements.

See also:

`numpy.fmax`

ProductSpaceUFuncs.fmin
`ProductSpaceUFuncs.`**`fmin`**(*x2*, *out=None*)
Element-wise minimum of array elements.

See also:

`numpy.fmin`

ProductSpaceUFuncs.fmod
`ProductSpaceUFuncs.`**`fmod`**(*x2*, *out=None*)
Return the element-wise remainder of division.

See also:

`numpy.fmod`

ProductSpaceUFuncs.greater
`ProductSpaceUFuncs.`**`greater`**(*x2*, *out=None*)
Return the truth value of (x1 > x2) element-wise.

See also:

`numpy.greater`

ProductSpaceUFuncs.greater_equal
`ProductSpaceUFuncs.`**`greater_equal`**(*x2*, *out=None*)
Return the truth value of (x1 >= x2) element-wise.

See also:

`numpy.greater_equal`

ProductSpaceUFuncs.hypot
ProductSpaceUFuncs.**hypot**(*x2*, *out=None*)
Given the "legs" of a right triangle, return its hypotenuse.

See also:

numpy.hypot

ProductSpaceUFuncs.invert
ProductSpaceUFuncs.**invert**(*out=None*)
Compute bit-wise inversion, or bit-wise NOT, element-wise.

See also:

numpy.invert

ProductSpaceUFuncs.isfinite
ProductSpaceUFuncs.**isfinite**(*out=None*)
Test element-wise for finiteness (not infinity or not Not a Number).

See also:

numpy.isfinite

ProductSpaceUFuncs.isinf
ProductSpaceUFuncs.**isinf**(*out=None*)
Test element-wise for positive or negative infinity.

See also:

numpy.isinf

ProductSpaceUFuncs.isnan
ProductSpaceUFuncs.**isnan**(*out=None*)
Test element-wise for NaN and return result as a boolean array.

See also:

numpy.isnan

ProductSpaceUFuncs.left_shift
ProductSpaceUFuncs.**left_shift**(*x2*, *out=None*)
Shift the bits of an integer to the left.

See also:

numpy.left_shift

ProductSpaceUFuncs.less
ProductSpaceUFuncs.**less**(*x2*, *out=None*)
Return the truth value of (x1 < x2) element-wise.

See also:

numpy.less

ProductSpaceUFuncs.less_equal
`ProductSpaceUFuncs.`**`less_equal`**(*x2*, *out=None*)
Return the truth value of (x1 =< x2) element-wise.

See also:

`numpy.less_equal`

ProductSpaceUFuncs.log
`ProductSpaceUFuncs.`**`log`**(*out=None*)
Natural logarithm, element-wise.

See also:

`numpy.log`

ProductSpaceUFuncs.log10
`ProductSpaceUFuncs.`**`log10`**(*out=None*)
Return the base 10 logarithm of the input array, element-wise.

See also:

`numpy.log10`

ProductSpaceUFuncs.log1p
`ProductSpaceUFuncs.`**`log1p`**(*out=None*)
Return the natural logarithm of one plus the input array, element-wise.

See also:

`numpy.log1p`

ProductSpaceUFuncs.log2
`ProductSpaceUFuncs.`**`log2`**(*out=None*)
Base-2 logarithm of x.

See also:

`numpy.log2`

ProductSpaceUFuncs.logaddexp
`ProductSpaceUFuncs.`**`logaddexp`**(*x2*, *out=None*)
Logarithm of the sum of exponentiations of the inputs.

See also:

`numpy.logaddexp`

ProductSpaceUFuncs.logaddexp2
`ProductSpaceUFuncs.`**`logaddexp2`**(*x2*, *out=None*)
Logarithm of the sum of exponentiations of the inputs in base-2.

See also:

`numpy.logaddexp2`

ProductSpaceUFuncs.logical_and
`ProductSpaceUFuncs.`**`logical_and`**(*x2*, *out=None*)
Compute the truth value of x1 AND x2 element-wise.

See also:

`numpy.logical_and`

ProductSpaceUFuncs.logical_not
`ProductSpaceUFuncs.`**`logical_not`**(*out=None*)
Compute the truth value of NOT x element-wise.

See also:

`numpy.logical_not`

ProductSpaceUFuncs.logical_or
`ProductSpaceUFuncs.`**`logical_or`**(*x2*, *out=None*)
Compute the truth value of x1 OR x2 element-wise.

See also:

`numpy.logical_or`

ProductSpaceUFuncs.logical_xor
`ProductSpaceUFuncs.`**`logical_xor`**(*x2*, *out=None*)
Compute the truth value of x1 XOR x2, element-wise.

See also:

`numpy.logical_xor`

ProductSpaceUFuncs.max
`ProductSpaceUFuncs.`**`max`**()
Maximum value in array.

See also:

`numpy.max`

ProductSpaceUFuncs.maximum
`ProductSpaceUFuncs.`**`maximum`**(*x2*, *out=None*)
Element-wise maximum of array elements.

See also:

`numpy.maximum`

ProductSpaceUFuncs.min
`ProductSpaceUFuncs.`**`min`**()
Minimum value in array.

See also:

`numpy.min`

ProductSpaceUFuncs.minimum
`ProductSpaceUFuncs.`**`minimum`**(*x2*, *out=None*)
Element-wise minimum of array elements.

See also:

`numpy.minimum`

ProductSpaceUFuncs.mod
`ProductSpaceUFuncs.`**`mod`**(*x2*, *out=None*)
Return element-wise remainder of division.

See also:

`numpy.mod`

ProductSpaceUFuncs.modf
`ProductSpaceUFuncs.`**`modf`**(*out1=None*, *out2=None*)
Return the fractional and integral parts of an array, element-wise.

See also:

`numpy.modf`

ProductSpaceUFuncs.multiply
`ProductSpaceUFuncs.`**`multiply`**(*x2*, *out=None*)
Multiply arguments element-wise with numpy.

See also:

`numpy.multiply`

ProductSpaceUFuncs.negative
`ProductSpaceUFuncs.`**`negative`**(*out=None*)
Numerical negative, element-wise.

See also:

`numpy.negative`

ProductSpaceUFuncs.nextafter
`ProductSpaceUFuncs.`**`nextafter`**(*x2*, *out=None*)
Return the next floating-point value after x1 towards x2, element-wise.

See also:

`numpy.nextafter`

ProductSpaceUFuncs.not_equal
`ProductSpaceUFuncs.`**`not_equal`**(*x2*, *out=None*)
Return (x1 != x2) element-wise.

See also:

`numpy.not_equal`

ProductSpaceUFuncs.power
`ProductSpaceUFuncs.`**`power`**(*x2*, *out=None*)
First array elements raised to powers from second array, element-wise.

See also:

`numpy.power`

ProductSpaceUFuncs.prod
`ProductSpaceUFuncs.`**`prod`**()
Product of array elements.

See also:

`numpy.prod`

ProductSpaceUFuncs.rad2deg
`ProductSpaceUFuncs.`**`rad2deg`**(*out=None*)
Convert angles from radians to degrees.

See also:

`numpy.rad2deg`

ProductSpaceUFuncs.reciprocal
`ProductSpaceUFuncs.`**`reciprocal`**(*out=None*)
Return the reciprocal of the argument, element-wise.

See also:

`numpy.reciprocal`

ProductSpaceUFuncs.remainder
`ProductSpaceUFuncs.`**`remainder`**(*x2*, *out=None*)
Return element-wise remainder of division.

See also:

`numpy.remainder`

ProductSpaceUFuncs.right_shift
`ProductSpaceUFuncs.`**`right_shift`**(*x2*, *out=None*)
Shift the bits of an integer to the right.

See also:

`numpy.right_shift`

ProductSpaceUFuncs.rint
`ProductSpaceUFuncs.`**`rint`**(*out=None*)
Round elements of the array to the nearest integer.

See also:

`numpy.rint`

ProductSpaceUFuncs.sign
`ProductSpaceUFuncs.`**`sign`**(*out=None*)
Returns an element-wise indication of the sign of a number.

See also:

`numpy.sign`

ProductSpaceUFuncs.signbit
`ProductSpaceUFuncs.`**`signbit`**(*out=None*)
Returns element-wise True where signbit is set (less than zero).

See also:

`numpy.signbit`

ProductSpaceUFuncs.sin
`ProductSpaceUFuncs.`**`sin`**(*out=None*)
Trigonometric sine, element-wise.

See also:

`numpy.sin`

ProductSpaceUFuncs.sinh
`ProductSpaceUFuncs.`**`sinh`**(*out=None*)
Hyperbolic sine, element-wise.

See also:

`numpy.sinh`

ProductSpaceUFuncs.sqrt
`ProductSpaceUFuncs.`**`sqrt`**(*out=None*)
Return the positive square-root of an array, element-wise.

See also:

`numpy.sqrt`

ProductSpaceUFuncs.square
`ProductSpaceUFuncs.`**`square`**(*out=None*)
Return the element-wise square of the input.

See also:

`numpy.square`

ProductSpaceUFuncs.subtract
`ProductSpaceUFuncs.`**`subtract`**(*x2*, *out=None*)
Subtract arguments, element-wise with numpy.

See also:

`numpy.subtract`

ProductSpaceUFuncs.sum
`ProductSpaceUFuncs.`**`sum`**`()`
Sum of array elements.

See also:

`numpy.sum`

ProductSpaceUFuncs.tan
`ProductSpaceUFuncs.`**`tan`**`(`*out=None*`)`
Compute tangent element-wise.

See also:

`numpy.tan`

ProductSpaceUFuncs.tanh
`ProductSpaceUFuncs.`**`tanh`**`(`*out=None*`)`
Compute hyperbolic tangent element-wise.

See also:

`numpy.tanh`

ProductSpaceUFuncs.true_divide
`ProductSpaceUFuncs.`**`true_divide`**`(`*x2*, *out=None*`)`
Returns a true division of the inputs, element-wise.

See also:

`numpy.true_divide`

ProductSpaceUFuncs.trunc
`ProductSpaceUFuncs.`**`trunc`**`(`*out=None*`)`
Return the truncated value of the input, element-wise.

See also:

`numpy.trunc`
`__init__``(`*vector*`)`
Create ufunc wrapper for vector.

Functions

method(self)	Maximum value in array.
wrap_reduction_base(name, descr)	Add ufunc methods to `NtuplesBaseVectorUFuncs`.
wrap_reduction_discretelp(name, descr)	
wrap_reduction_productspace(name, descr)	
wrap_ufunc_base(name, n_args, n_opt, descr)	Add ufunc methods to *NtuplesBaseUFuncs*.
wrap_ufunc_discretelp(name, n_args, n_opt, descr)	Add ufunc methods to *DiscreteLpUFuncs*.
wrap_ufunc_ntuples(name, n_args, n_opt, descr)	Add ufunc methods to *NtuplesUFuncs*.
wrap_ufunc_productspace(name, n_args, n_opt, ...)	Add ufunc methods to *ProductSpaceVector*.

method

odl.util.ufuncs.**method**(*self*)
Maximum value in array.

See also:

numpy.max

wrap_reduction_base
odl.util.ufuncs.**wrap_reduction_base**(*name*, *descr*)
Add ufunc methods to NtuplesBaseVectorUFuncs.

wrap_reduction_discretelp
odl.util.ufuncs.**wrap_reduction_discretelp**(*name*, *descr*)

wrap_reduction_productspace
odl.util.ufuncs.**wrap_reduction_productspace**(*name*, *descr*)

wrap_ufunc_base
odl.util.ufuncs.**wrap_ufunc_base**(*name*, *n_args*, *n_opt*, *descr*)
Add ufunc methods to *NtuplesBaseUFuncs*.

wrap_ufunc_discretelp
odl.util.ufuncs.**wrap_ufunc_discretelp**(*name*, *n_args*, *n_opt*, *descr*)
Add ufunc methods to *DiscreteLpUFuncs*.

wrap_ufunc_ntuples
odl.util.ufuncs.**wrap_ufunc_ntuples**(*name*, *n_args*, *n_opt*, *descr*)
Add ufunc methods to *NtuplesUFuncs*.

wrap_ufunc_productspace
odl.util.ufuncs.**wrap_ufunc_productspace**(*name*, *n_args*, *n_opt*, *descr*)
Add ufunc methods to *ProductSpaceVector*.

utility

Utilities for internal use.

Functions

array1d_repr(array)	Stringification of a 1D array, keeping byte / unicode.
array1d_str(array)	Stringification of a 1D array, regardless of byte or unicode.
arraynd_repr(array)	Stringification of an nD array, keeping byte / unicode.
arraynd_str(array)	Stringification of a nD array, regardless of byte or unicode.
dtype_repr(dtype)	Stringification of data type with default for int and float.
is_complex_floating_dtype(dtype)	Whether a NumPy datatype is complex floating-point or not.
is_floating_dtype(dtype)	Whether a NumPy datatype is floating or not.
is_real_dtype(dtype)	Whether a datatype is real (including integer) or not.

Continued on next page

Table 2.240 – continued from previous page

is_real_floating_dtype(dtype)	Whether a NumPy datatype is real floating-point or not.
is_scalar_dtype(dtype)	Whether a datatype is scalar or not.
with_metaclass(meta, *bases)	Function from jinja2/_compat.py.

array1d_repr

odl.util.utility.**array1d_repr**(*array*)
Stringification of a 1D array, keeping byte / unicode.

array1d_str

odl.util.utility.**array1d_str**(*array*)
Stringification of a 1D array, regardless of byte or unicode.

arraynd_repr

odl.util.utility.**arraynd_repr**(*array*)
Stringification of an nD array, keeping byte / unicode.

arraynd_str

odl.util.utility.**arraynd_str**(*array*)
Stringification of a nD array, regardless of byte or unicode.

dtype_repr

odl.util.utility.**dtype_repr**(*dtype*)
Stringification of data type with default for `int` and `float`.

is_complex_floating_dtype

odl.util.utility.**is_complex_floating_dtype**(*dtype*)
Whether a NumPy datatype is complex floating-point or not.

is_floating_dtype

odl.util.utility.**is_floating_dtype**(*dtype*)
Whether a NumPy datatype is floating or not.

is_real_dtype

odl.util.utility.**is_real_dtype**(*dtype*)
Whether a datatype is real (including integer) or not.

is_real_floating_dtype

odl.util.utility.**is_real_floating_dtype**(*dtype*)
Whether a NumPy datatype is real floating-point or not.

is_scalar_dtype

odl.util.utility.**is_scalar_dtype**(*dtype*)
Whether a datatype is scalar or not.

with_metaclass

odl.util.utility.**with_metaclass**(*meta*, **bases*)
Function from jinja2/_compat.py. License: BSD.

Use it like this:

```
class BaseForm(object):
    pass

class FormType(type):
    pass

class Form(with_metaclass(FormType, BaseForm)):
    pass
```

This requires a bit of explanation: the basic idea is to make a dummy metaclass for one level of class instantiation that replaces itself with the actual metaclass. Because of internal type checks we also need to make sure that we downgrade the custom metaclass for one level to something closer to type (that's why __call__ and __init__ comes back from type etc.).

This has the advantage over six.with_metaclass of not introducing dummy classes into the final MRO.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

[R1] Griva, Igor, Stephen G. Nash, and Ariela Sofer. Linear and nonlinear optimization. Siam, 2009

[R2] Broyden, Charles G. “A class of methods for solving nonlinear simultaneous equations.” Mathematics of computation (1965): 577-593.

[R3] Kvaalen, Eric. “A faster Broyden method.” BIT Numerical Mathematics 31.2 (1991): 369-372.

[R4] Broyden, Charles G. “A class of methods for solving nonlinear simultaneous equations.” Mathematics of computation (1965): 577-593.

[R5] Kvaalen, Eric. “A faster Broyden method.” BIT Numerical Mathematics 31.2 (1991): 369-372.

[R6] Boyd, Stephen, and Lieven Vandenberghe. Convex optimization. Cambridge university press, 2004. Available at

[R7] Griva, Igor, Stephen G. Nash, and Ariela Sofer. Linear and nonlinear optimization. Siam, 2009

[R8] Boyd, Stephen, and Lieven Vandenberghe. Convex optimization. Cambridge university press, 2004. Available at

[R9] Pages 378-379 in Griva, Igor, Stephen G. Nash, and Ariela Sofer. Linear and nonlinear optimization. Siam, 2009.

[R10] J. Barzilai and J. M. Borwein. Two-point step size gradient methods. IMA Journal of Numerical Analysis, vol. 8, pp. 141–148, 1988.

[R11] M. Raydan. The Barzilai and Borwein method for the large scale unconstrained minimization problem. SIAM J. Optim., vol. 7, pp. 26–33, 1997.

[R12] Boyd, Stephen, and Lieven Vandenberghe. Convex optimization. Cambridge university press, 2004.

[R13] Griva, Igor, Stephen G. Nash, and Ariela Sofer. Linear and nonlinear optimization. Siam, 2009

Symbols

__call__() (odl.operator.operator.FunctionalLeftVectorMult method), 122
__call__() (odl.operator.operator.Operator method), 126
__call__() (odl.operator.operator.OperatorComp method), 128
__call__() (odl.operator.operator.OperatorLeftScalarMult method), 131
__call__() (odl.operator.operator.OperatorLeftVectorMult method), 134
__call__() (odl.operator.operator.OperatorPointwiseProduct method), 136
__call__() (odl.operator.operator.OperatorRightScalarMult method), 139
__call__() (odl.operator.operator.OperatorRightVectorMult method), 141
__call__() (odl.operator.operator.OperatorSum method), 144
__call__() (odl.solvers.scalar.steplen.BacktrackingLineSearch method), 201
__call__() (odl.solvers.scalar.steplen.BarzilaiBorweinStep method), 202
__call__() (odl.solvers.scalar.steplen.ConstantLineSearch method), 203
__call__() (odl.solvers.scalar.steplen.LineSearch method), 204
__call__() (odl.solvers.scalar.steplen.StepLength method), 204
__call__() (odl.space.fspace.FunctionSetVector method), 254
__call__() (odl.space.fspace.FunctionSpaceVector method), 259
__contains__() (odl.discr.grid.RegularGrid method), 76
__contains__() (odl.discr.grid.TensorGrid method), 84
__contains__() (odl.set.domain.IntervalProd method), 159
__contains__() (odl.set.sets.CartesianProduct method), 173
__contains__() (odl.set.sets.ComplexNumbers method), 174
__contains__() (odl.set.sets.EmptySet method), 175
__contains__() (odl.set.sets.Field method), 176
__contains__() (odl.set.sets.Integers method), 177
__contains__() (odl.set.sets.RealNumbers method), 178
__contains__() (odl.set.sets.Set method), 179
__contains__() (odl.set.sets.Strings method), 180
__contains__() (odl.set.sets.UniversalSet method), 181
__contains__() (odl.set.space.LinearSpace method), 183
__contains__() (odl.set.space.UniversalSpace method), 190
__contains__() (odl.space.base_ntuples.FnBase method), 209
__contains__() (odl.space.base_ntuples.NtuplesBase method), 216
__contains__() (odl.space.fspace.FunctionSet method), 252
__contains__() (odl.space.fspace.FunctionSpace method), 256
__eq__() (odl.discr.discr_mappings.FunctionSetMapping method), 40
__eq__() (odl.discr.discr_mappings.GridCollocation method), 41
__eq__() (odl.discr.discr_mappings.LinearInterpolation method), 44
__eq__() (odl.discr.discr_mappings.NearestInterpolation method), 46
__eq__() (odl.discr.discr_mappings.RawGridCollocation method), 49
__eq__() (odl.discr.discr_mappings.RawNearestInterpolation method), 52
__eq__() (odl.discr.discretization.Discretization method), 64
__eq__() (odl.discr.discretization.DiscretizationVector method), 67
__eq__() (odl.discr.discretization.RawDiscretization method), 70
__eq__() (odl.discr.discretization.RawDiscretizationVector method), 72
__eq__() (odl.discr.grid.RegularGrid method), 76
__eq__() (odl.discr.grid.TensorGrid method), 84
__eq__() (odl.set.domain.IntervalProd method), 159

__eq__() (odl.set.pspace.ProductSpace method), 166
__eq__() (odl.set.pspace.ProductSpaceVector method), 171
__eq__() (odl.set.sets.CartesianProduct method), 173
__eq__() (odl.set.sets.ComplexNumbers method), 174
__eq__() (odl.set.sets.EmptySet method), 175
__eq__() (odl.set.sets.Field method), 176
__eq__() (odl.set.sets.Integers method), 177
__eq__() (odl.set.sets.RealNumbers method), 178
__eq__() (odl.set.sets.Set method), 179
__eq__() (odl.set.sets.Strings method), 180
__eq__() (odl.set.sets.UniversalSet method), 181
__eq__() (odl.set.space.LinearSpaceVector method), 188
__eq__() (odl.set.space.UniversalSpace method), 191
__eq__() (odl.space.base_ntuples.FnBase method), 210
__eq__() (odl.space.base_ntuples.FnBaseVector method), 213
__eq__() (odl.space.base_ntuples.FnWeightingBase method), 214
__eq__() (odl.space.base_ntuples.NtuplesBase method), 217
__eq__() (odl.space.base_ntuples.NtuplesBaseVector method), 219
__eq__() (odl.space.cu_ntuples.CudaFn method), 222
__eq__() (odl.space.cu_ntuples.CudaFnConstWeighting method), 228
__eq__() (odl.space.cu_ntuples.CudaFnCustomDist method), 230
__eq__() (odl.space.cu_ntuples.CudaFnCustomInnerProduct method), 231
__eq__() (odl.space.cu_ntuples.CudaFnCustomNorm method), 232
__eq__() (odl.space.cu_ntuples.CudaFnNoWeighting method), 234
__eq__() (odl.space.cu_ntuples.CudaFnVectorWeighting method), 241
__eq__() (odl.space.cu_ntuples.CudaNtuplesVector method), 246
__eq__() (odl.space.fspace.FunctionSet method), 252
__eq__() (odl.space.fspace.FunctionSetVector method), 254
__eq__() (odl.space.fspace.FunctionSpace method), 256
__eq__() (odl.space.fspace.FunctionSpaceVector method), 259
__eq__() (odl.space.ntuples.Fn method), 262
__eq__() (odl.space.ntuples.FnConstWeighting method), 269
__eq__() (odl.space.ntuples.FnCustomDist method), 271
__eq__() (odl.space.ntuples.FnCustomInnerProduct method), 272
__eq__() (odl.space.ntuples.FnCustomNorm method), 274
__eq__() (odl.space.ntuples.FnMatrixWeighting method), 275
__eq__() (odl.space.ntuples.FnNoWeighting method), 278
__eq__() (odl.space.ntuples.FnVectorWeighting method), 286
__eq__() (odl.space.ntuples.NtuplesVector method), 293
__getitem__() (odl.discr.discretization.DiscretizationVector method), 67
__getitem__() (odl.discr.discretization.RawDiscretizationVector method), 72
__getitem__() (odl.discr.grid.RegularGrid method), 76
__getitem__() (odl.discr.grid.TensorGrid method), 85
__getitem__() (odl.set.pspace.ProductSpace method), 167
__getitem__() (odl.set.pspace.ProductSpaceVector method), 172
__getitem__() (odl.set.sets.CartesianProduct method), 173
__getitem__() (odl.space.base_ntuples.FnBaseVector method), 213
__getitem__() (odl.space.base_ntuples.NtuplesBaseVector method), 219
__getitem__() (odl.space.cu_ntuples.CudaFnVector method), 238
__getitem__() (odl.space.cu_ntuples.CudaNtuplesVector method), 247
__getitem__() (odl.space.ntuples.FnVector method), 282
__getitem__() (odl.space.ntuples.NtuplesVector method), 294
__init__() (odl.diagnostics.operator.OperatorTest method), 34
__init__() (odl.diagnostics.space.SpaceTest method), 38
__init__() (odl.discr.discr_mappings.FunctionSetMapping method), 40
__init__() (odl.discr.discr_mappings.GridCollocation method), 42
__init__() (odl.discr.discr_mappings.LinearInterpolation method), 44
__init__() (odl.discr.discr_mappings.NearestInterpolation method), 47
__init__() (odl.discr.discr_mappings.RawGridCollocation method), 50
__init__() (odl.discr.discr_mappings.RawNearestInterpolation method), 53
__init__() (odl.discr.discr_ops.DiscreteDivergence method), 55
__init__() (odl.discr.discr_ops.DiscreteGradient method), 58
__init__() (odl.discr.discr_ops.DiscretePartDeriv method), 60
__init__() (odl.discr.discretization.Discretization method), 65
__init__() (odl.discr.discretization.DiscretizationVector method), 68

__init__() (odl.discr.discretization.RawDiscretization method), 70
__init__() (odl.discr.discretization.RawDiscretizationVector method), 73
__init__() (odl.discr.grid.RegularGrid method), 82
__init__() (odl.discr.grid.TensorGrid method), 90
__init__() (odl.discr.lp_discr.DiscreteLp method), 95
__init__() (odl.discr.lp_discr.DiscreteLpVector method), 99
__init__() (odl.operator.default_ops.ConstantOperator method), 104
__init__() (odl.operator.default_ops.IdentityOperator method), 106
__init__() (odl.operator.default_ops.InnerProductAdjointOperator method), 108
__init__() (odl.operator.default_ops.InnerProductOperator method), 110
__init__() (odl.operator.default_ops.LinCombOperator method), 112
__init__() (odl.operator.default_ops.MultiplyOperator method), 114
__init__() (odl.operator.default_ops.ResidualOperator method), 116
__init__() (odl.operator.default_ops.ScalingOperator method), 118
__init__() (odl.operator.default_ops.ZeroOperator method), 120
__init__() (odl.operator.operator.FunctionalLeftVectorMult method), 123
__init__() (odl.operator.operator.Operator method), 127
__init__() (odl.operator.operator.OperatorComp method), 129
__init__() (odl.operator.operator.OperatorLeftScalarMult method), 132
__init__() (odl.operator.operator.OperatorLeftVectorMult method), 135
__init__() (odl.operator.operator.OperatorPointwiseProduct method), 137
__init__() (odl.operator.operator.OperatorRightScalarMult method), 140
__init__() (odl.operator.operator.OperatorRightVectorMult method), 142
__init__() (odl.operator.operator.OperatorSum method), 145
__init__() (odl.operator.pspace_ops.ComponentProjection method), 150
__init__() (odl.operator.pspace_ops.ComponentProjectionAdjoint method), 153
__init__() (odl.operator.pspace_ops.ProductSpaceOperator method), 156
__init__() (odl.set.domain.IntervalProd method), 164
__init__() (odl.set.pspace.ProductSpace method), 168
__init__() (odl.set.pspace.ProductSpaceVector method), 172
__init__() (odl.set.sets.CartesianProduct method), 174
__init__() (odl.set.sets.Strings method), 181
__init__() (odl.set.space.LinearSpace method), 186
__init__() (odl.set.space.LinearSpaceVector method), 189
__init__() (odl.set.space.UniversalSpace method), 194
__init__() (odl.solvers.scalar.steplen.BacktrackingLineSearch method), 202
__init__() (odl.solvers.scalar.steplen.BarzilaiBorweinStep method), 203
__init__() (odl.solvers.scalar.steplen.ConstantLineSearch method), 203
__init__() (odl.solvers.util.partial.ForEachPartial method), 205
__init__() (odl.solvers.util.partial.PrintIterationPartial method), 206
__init__() (odl.solvers.util.partial.PrintStatusPartial method), 206
__init__() (odl.solvers.util.partial.StorePartial method), 207
__init__() (odl.space.base_ntuples.FnBase method), 211
__init__() (odl.space.base_ntuples.FnBaseVector method), 214
__init__() (odl.space.base_ntuples.FnWeightingBase method), 215
__init__() (odl.space.base_ntuples.NtuplesBase method), 217
__init__() (odl.space.base_ntuples.NtuplesBaseVector method), 220
__init__() (odl.space.cu_ntuples.CudaFn method), 226
__init__() (odl.space.cu_ntuples.CudaFnConstWeighting method), 229
__init__() (odl.space.cu_ntuples.CudaFnCustomDist method), 231
__init__() (odl.space.cu_ntuples.CudaFnCustomInnerProduct method), 232
__init__() (odl.space.cu_ntuples.CudaFnCustomNorm method), 233
__init__() (odl.space.cu_ntuples.CudaFnNoWeighting method), 235
__init__() (odl.space.cu_ntuples.CudaFnVector method), 240
__init__() (odl.space.cu_ntuples.CudaFnVectorWeighting method), 242
__init__() (odl.space.cu_ntuples.CudaNtuples method), 243
__init__() (odl.space.cu_ntuples.CudaNtuplesVector method), 249
__init__() (odl.space.fspace.FunctionSet method), 252
__init__() (odl.space.fspace.FunctionSetVector method), 254
__init__() (odl.space.fspace.FunctionSpace method), 257
__init__() (odl.space.fspace.FunctionSpaceVector method), 260
__init__() (odl.space.ntuples.Fn method), 267

__init__() (odl.space.ntuples.FnConstWeighting method), 270
__init__() (odl.space.ntuples.FnCustomDist method), 272
__init__() (odl.space.ntuples.FnCustomInnerProduct method), 273
__init__() (odl.space.ntuples.FnCustomNorm method), 274
__init__() (odl.space.ntuples.FnMatrixWeighting method), 276
__init__() (odl.space.ntuples.FnNoWeighting method), 278
__init__() (odl.space.ntuples.FnVector method), 285
__init__() (odl.space.ntuples.FnVectorWeighting method), 287
__init__() (odl.space.ntuples.MatVecOperator method), 289
__init__() (odl.space.ntuples.Ntuples method), 291
__init__() (odl.space.ntuples.NtuplesVector method), 296
__init__() (odl.tomo.backends.stir_bindings.BackProjectorByBinWrapper method), 301
__init__() (odl.tomo.backends.stir_bindings.ForwardProjectorByBinWrapper method), 303
__init__() (odl.tomo.backends.stir_bindings.StirVerbosity method), 304
__init__() (odl.trafos.wavelet.DiscreteWaveletTransform method), 307
__init__() (odl.trafos.wavelet.DiscreteWaveletTransformInverse method), 309
__init__() (odl.util.testutils.FailCounter method), 316
__init__() (odl.util.testutils.ProgressBar method), 317
__init__() (odl.util.testutils.ProgressRange method), 317
__init__() (odl.util.testutils.Timer method), 318
__init__() (odl.util.ufuncs.CudaNtuplesUFuncs method), 332
__init__() (odl.util.ufuncs.DiscreteLpUFuncs method), 345
__init__() (odl.util.ufuncs.NtuplesBaseUFuncs method), 358
__init__() (odl.util.ufuncs.NtuplesUFuncs method), 370
__init__() (odl.util.ufuncs.ProductSpaceUFuncs method), 383
__setitem__() (odl.discr.discretization.DiscretizationVector method), 68
__setitem__() (odl.discr.discretization.RawDiscretizationVector method), 72
__setitem__() (odl.discr.lp_discr.DiscreteLpVector method), 98
__setitem__() (odl.set.pspace.ProductSpaceVector method), 172
__setitem__() (odl.space.base_ntuples.FnBaseVector method), 213
__setitem__() (odl.space.base_ntuples.NtuplesBaseVector method), 219
__setitem__() (odl.space.cu_ntuples.CudaFnVector method), 238
__setitem__() (odl.space.cu_ntuples.CudaNtuplesVector method), 247
__setitem__() (odl.space.ntuples.FnVector method), 283
__setitem__() (odl.space.ntuples.NtuplesVector method), 294
_apply() (odl.discr.discr_ops.DiscreteDivergence method), 55
_apply() (odl.discr.discr_ops.DiscreteGradient method), 58
_apply() (odl.discr.discr_ops.DiscretePartDeriv method), 59
_apply() (odl.operator.default_ops.ConstantOperator method), 103
_apply() (odl.operator.default_ops.IdentityOperator method), 106
_apply() (odl.operator.default_ops.InnerProductAdjointOperator method), 108
_apply() (odl.operator.default_ops.LinCombOperator method), 111
_apply() (odl.operator.default_ops.MultiplyOperator method), 113
_apply() (odl.operator.default_ops.ResidualOperator method), 115
_apply() (odl.operator.default_ops.ScalingOperator method), 117
_apply() (odl.operator.default_ops.ZeroOperator method), 120
_apply() (odl.operator.operator.FunctionalLeftVectorMult method), 123
_apply() (odl.operator.operator.OperatorComp method), 129
_apply() (odl.operator.operator.OperatorLeftScalarMult method), 132
_apply() (odl.operator.operator.OperatorLeftVectorMult method), 134
_apply() (odl.operator.operator.OperatorPointwiseProduct method), 137
_apply() (odl.operator.operator.OperatorRightScalarMult method), 139
_apply() (odl.operator.operator.OperatorRightVectorMult method), 142
_apply() (odl.operator.operator.OperatorSum method), 144
_apply() (odl.operator.pspace_ops.ComponentProjection method), 149
_apply() (odl.operator.pspace_ops.ComponentProjectionAdjoint method), 152
_apply() (odl.operator.pspace_ops.ProductSpaceOperator method), 155
_apply() (odl.space.ntuples.MatVecOperator method), 289

_apply() (odl.tomo.backends.stir_bindings.BackProjectorByBinWrapper method), 301
_apply() (odl.tomo.backends.stir_bindings.ForwardProjectorByBinWrapper method), 303
_call() (odl.discr.discr_mappings.GridCollocation method), 41
_call() (odl.discr.discr_mappings.LinearInterpolation method), 44
_call() (odl.discr.discr_mappings.NearestInterpolation method), 46
_call() (odl.discr.discr_mappings.RawGridCollocation method), 49
_call() (odl.discr.discr_mappings.RawNearestInterpolation method), 52
_call() (odl.operator.default_ops.ConstantOperator method), 104
_call() (odl.operator.default_ops.IdentityOperator method), 106
_call() (odl.operator.default_ops.InnerProductAdjointOperator method), 108
_call() (odl.operator.default_ops.InnerProductOperator method), 110
_call() (odl.operator.default_ops.ResidualOperator method), 115
_call() (odl.operator.default_ops.ScalingOperator method), 118
_call() (odl.operator.default_ops.ZeroOperator method), 120
_call() (odl.operator.operator.FunctionalLeftVectorMult method), 123
_call() (odl.operator.operator.OperatorComp method), 129
_call() (odl.operator.operator.OperatorLeftScalarMult method), 132
_call() (odl.operator.operator.OperatorLeftVectorMult method), 134
_call() (odl.operator.operator.OperatorPointwiseProduct method), 137
_call() (odl.operator.operator.OperatorRightScalarMult method), 139
_call() (odl.operator.operator.OperatorRightVectorMult method), 142
_call() (odl.operator.operator.OperatorSum method), 145
_call() (odl.operator.pspace_ops.ComponentProjection method), 149
_call() (odl.operator.pspace_ops.ComponentProjectionAdjoint method), 152
_call() (odl.operator.pspace_ops.ProductSpaceOperator method), 155
_call() (odl.space.ntuples.MatVecOperator method), 289
_call() (odl.trafos.wavelet.DiscreteWaveletTransform method), 307
_call() (odl.trafos.wavelet.DiscreteWaveletTransformInverse method), 308
_divide() (odl.discr.discretization.Discretization method), 64
_divide() (odl.set.pspace.ProductSpace method), 167
_dist() (odl.set.space.LinearSpace method), 183
_dist() (odl.set.space.UniversalSpace method), 191
_dist() (odl.space.cu_ntuples.CudaFn method), 223
_dist() (odl.space.ntuples.Fn method), 263
_divide() (odl.discr.discretization.Discretization method), 64
_divide() (odl.set.space.UniversalSpace method), 191
_divide() (odl.space.base_ntuples.FnBase method), 210
_divide() (odl.space.cu_ntuples.CudaFn method), 223
_divide() (odl.space.ntuples.Fn method), 263
_inner() (odl.discr.discretization.Discretization method), 64
_inner() (odl.set.pspace.ProductSpace method), 167
_inner() (odl.set.space.LinearSpace method), 183
_inner() (odl.set.space.UniversalSpace method), 191
_inner() (odl.space.cu_ntuples.CudaFn method), 223
_inner() (odl.space.ntuples.Fn method), 264
_lincomb() (odl.diagnostics.space.SpaceTest method), 35
_lincomb() (odl.discr.discretization.Discretization method), 64
_lincomb() (odl.set.pspace.ProductSpace method), 167
_lincomb() (odl.set.space.LinearSpace method), 183
_lincomb() (odl.set.space.UniversalSpace method), 191
_lincomb() (odl.space.cu_ntuples.CudaFn method), 224
_lincomb() (odl.space.fspace.FunctionSpace method), 256
_lincomb() (odl.space.ntuples.Fn method), 264
_multiply() (odl.discr.discretization.Discretization method), 65
_multiply() (odl.set.pspace.ProductSpace method), 167
_multiply() (odl.set.space.LinearSpace method), 183
_multiply() (odl.set.space.UniversalSpace method), 191
_multiply() (odl.space.base_ntuples.FnBase method), 211
_multiply() (odl.space.cu_ntuples.CudaFn method), 224
_multiply() (odl.space.fspace.FunctionSpace method), 256
_multiply() (odl.space.ntuples.Fn method), 265
_norm() (odl.discr.discretization.Discretization method), 65
_norm() (odl.set.pspace.ProductSpace method), 167
_norm() (odl.set.space.LinearSpace method), 184
_norm() (odl.set.space.UniversalSpace method), 191
_norm() (odl.space.cu_ntuples.CudaFn method), 225
_norm() (odl.space.ntuples.Fn method), 265

A

absolute() (odl.util.ufuncs.CudaNtuplesUFuncs method), 321
absolute() (odl.util.ufuncs.DiscreteLpUFuncs method), 334

absolute() (odl.util.ufuncs.NtuplesBaseUFuncs method), 347
absolute() (odl.util.ufuncs.NtuplesUFuncs method), 359
absolute() (odl.util.ufuncs.ProductSpaceUFuncs method), 372
add() (odl.util.ufuncs.CudaNtuplesUFuncs method), 321
add() (odl.util.ufuncs.DiscreteLpUFuncs method), 334
add() (odl.util.ufuncs.NtuplesBaseUFuncs method), 347
add() (odl.util.ufuncs.NtuplesUFuncs method), 360
add() (odl.util.ufuncs.ProductSpaceUFuncs method), 372
adjoint (odl.discr.discr_mappings.FunctionSetMapping attribute), 39
adjoint (odl.discr.discr_mappings.GridCollocation attribute), 40
adjoint (odl.discr.discr_mappings.LinearInterpolation attribute), 43
adjoint (odl.discr.discr_mappings.NearestInterpolation attribute), 45
adjoint (odl.discr.discr_mappings.RawGridCollocation attribute), 48
adjoint (odl.discr.discr_mappings.RawNearestInterpolation attribute), 51
adjoint (odl.discr.discr_ops.DiscreteDivergence attribute), 54
adjoint (odl.discr.discr_ops.DiscreteGradient attribute), 56
adjoint (odl.discr.discr_ops.DiscretePartDeriv attribute), 59
adjoint (odl.operator.default_ops.ConstantOperator attribute), 103
adjoint (odl.operator.default_ops.IdentityOperator attribute), 105
adjoint (odl.operator.default_ops.InnerProductAdjointOperator attribute), 107
adjoint (odl.operator.default_ops.InnerProductOperator attribute), 109
adjoint (odl.operator.default_ops.LinCombOperator attribute), 110
adjoint (odl.operator.default_ops.MultiplyOperator attribute), 112
adjoint (odl.operator.default_ops.ResidualOperator attribute), 114
adjoint (odl.operator.default_ops.ScalingOperator attribute), 116
adjoint (odl.operator.default_ops.ZeroOperator attribute), 119
adjoint (odl.operator.operator.FunctionalLeftVectorMult attribute), 121
adjoint (odl.operator.operator.Operator attribute), 125
adjoint (odl.operator.operator.OperatorComp attribute), 127
adjoint (odl.operator.operator.OperatorLeftScalarMult attribute), 130
adjoint (odl.operator.operator.OperatorLeftVectorMult attribute), 133
adjoint (odl.operator.operator.OperatorPointwiseProduct attribute), 135
adjoint (odl.operator.operator.OperatorRightScalarMult attribute), 138
adjoint (odl.operator.operator.OperatorRightVectorMult attribute), 140
adjoint (odl.operator.operator.OperatorSum attribute), 143
adjoint (odl.operator.pspace_ops.ComponentProjection attribute), 148
adjoint (odl.operator.pspace_ops.ComponentProjectionAdjoint attribute), 151
adjoint (odl.operator.pspace_ops.ProductSpaceOperator attribute), 154
adjoint (odl.space.fspace.FunctionSetVector attribute), 253
adjoint (odl.space.fspace.FunctionSpaceVector attribute), 258
adjoint (odl.space.ntuples.MatVecOperator attribute), 288
adjoint (odl.tomo.backends.stir_bindings.BackProjectorByBinWrapper attribute), 301
adjoint (odl.tomo.backends.stir_bindings.ForwardProjectorByBinWrapper attribute), 302
adjoint (odl.trafos.wavelet.DiscreteWaveletTransform attribute), 306
adjoint (odl.trafos.wavelet.DiscreteWaveletTransformInverse attribute), 308
adjoint() (odl.diagnostics.operator.OperatorTest method), 33
all_almost_equal() (in module odl.util.testutils), 318
all_equal() (in module odl.util.testutils), 318
almost_equal() (in module odl.util.testutils), 318
apply() (odl.space.fspace.FunctionSetVector method), 254
apply() (odl.space.fspace.FunctionSpaceVector method), 259
approx_contains() (odl.discr.grid.RegularGrid method), 77
approx_contains() (odl.discr.grid.TensorGrid method), 85
approx_contains() (odl.set.domain.IntervalProd method), 159
approx_equals() (odl.discr.grid.RegularGrid method), 77
approx_equals() (odl.discr.grid.TensorGrid method), 85
approx_equals() (odl.set.domain.IntervalProd method), 160
arccos() (odl.util.ufuncs.CudaNtuplesUFuncs method), 321
arccos() (odl.util.ufuncs.DiscreteLpUFuncs method), 334
arccos() (odl.util.ufuncs.NtuplesBaseUFuncs method), 347
arccos() (odl.util.ufuncs.NtuplesUFuncs method), 360

arccos() (odl.util.ufuncs.ProductSpaceUFuncs method), 373
arccosh() (odl.util.ufuncs.CudaNtuplesUFuncs method), 321
arccosh() (odl.util.ufuncs.DiscreteLpUFuncs method), 334
arccosh() (odl.util.ufuncs.NtuplesBaseUFuncs method), 347
arccosh() (odl.util.ufuncs.NtuplesUFuncs method), 360
arccosh() (odl.util.ufuncs.ProductSpaceUFuncs method), 373
arcsin() (odl.util.ufuncs.CudaNtuplesUFuncs method), 322
arcsin() (odl.util.ufuncs.DiscreteLpUFuncs method), 334
arcsin() (odl.util.ufuncs.NtuplesBaseUFuncs method), 347
arcsin() (odl.util.ufuncs.NtuplesUFuncs method), 360
arcsin() (odl.util.ufuncs.ProductSpaceUFuncs method), 373
arcsinh() (odl.util.ufuncs.CudaNtuplesUFuncs method), 322
arcsinh() (odl.util.ufuncs.DiscreteLpUFuncs method), 335
arcsinh() (odl.util.ufuncs.NtuplesBaseUFuncs method), 347
arcsinh() (odl.util.ufuncs.NtuplesUFuncs method), 360
arcsinh() (odl.util.ufuncs.ProductSpaceUFuncs method), 373
arctan() (odl.util.ufuncs.CudaNtuplesUFuncs method), 322
arctan() (odl.util.ufuncs.DiscreteLpUFuncs method), 335
arctan() (odl.util.ufuncs.NtuplesBaseUFuncs method), 348
arctan() (odl.util.ufuncs.NtuplesUFuncs method), 360
arctan() (odl.util.ufuncs.ProductSpaceUFuncs method), 373
arctan2() (odl.util.ufuncs.CudaNtuplesUFuncs method), 322
arctan2() (odl.util.ufuncs.DiscreteLpUFuncs method), 335
arctan2() (odl.util.ufuncs.NtuplesBaseUFuncs method), 348
arctan2() (odl.util.ufuncs.NtuplesUFuncs method), 361
arctan2() (odl.util.ufuncs.ProductSpaceUFuncs method), 373
arctanh() (odl.util.ufuncs.CudaNtuplesUFuncs method), 322
arctanh() (odl.util.ufuncs.DiscreteLpUFuncs method), 335
arctanh() (odl.util.ufuncs.NtuplesBaseUFuncs method), 348
arctanh() (odl.util.ufuncs.NtuplesUFuncs method), 361
arctanh() (odl.util.ufuncs.ProductSpaceUFuncs method), 373
area (odl.set.domain.IntervalProd attribute), 158
array1d_repr() (in module odl.util.utility), 385
array1d_str() (in module odl.util.utility), 385
array_to_pywt_coeff() (in module odl.trafos.wavelet), 309
arraynd_repr() (in module odl.util.utility), 385
arraynd_str() (in module odl.util.utility), 385
as_midp (odl.discr.grid.RegularGrid attribute), 74
as_midp (odl.discr.grid.TensorGrid attribute), 83
asarray() (odl.discr.discretization.DiscretizationVector method), 68
asarray() (odl.discr.discretization.RawDiscretizationVector method), 72
asarray() (odl.discr.lp_discr.DiscreteLpVector method), 99
asarray() (odl.space.base_ntuples.FnBaseVector method), 213
asarray() (odl.space.base_ntuples.NtuplesBaseVector method), 220
asarray() (odl.space.cu_ntuples.CudaFnVector method), 239
asarray() (odl.space.cu_ntuples.CudaNtuplesVector method), 248
asarray() (odl.space.ntuples.FnVector method), 284
asarray() (odl.space.ntuples.NtuplesVector method), 295
assign() (odl.set.space.LinearSpaceVector method), 189

B

BackProjectorByBinWrapper (class in odl.tomo.backends.stir_bindings), 300
BacktrackingLineSearch (class in odl.solvers.scalar.steplen), 201
BarzilaiBorweinStep (class in odl.solvers.scalar.steplen), 202
begin (odl.set.domain.IntervalProd attribute), 158
bfgs_method() (in module odl.solvers.findroot.newton), 194
bitwise_and() (odl.util.ufuncs.CudaNtuplesUFuncs method), 322
bitwise_and() (odl.util.ufuncs.DiscreteLpUFuncs method), 335
bitwise_and() (odl.util.ufuncs.NtuplesBaseUFuncs method), 348
bitwise_and() (odl.util.ufuncs.NtuplesUFuncs method), 361
bitwise_and() (odl.util.ufuncs.ProductSpaceUFuncs method), 374
bitwise_or() (odl.util.ufuncs.CudaNtuplesUFuncs method), 322
bitwise_or() (odl.util.ufuncs.DiscreteLpUFuncs method), 335
bitwise_or() (odl.util.ufuncs.NtuplesBaseUFuncs method), 348

bitwise_or() (odl.util.ufuncs.NtuplesUFuncs method), 361
bitwise_or() (odl.util.ufuncs.ProductSpaceUFuncs method), 374
bitwise_xor() (odl.util.ufuncs.CudaNtuplesUFuncs method), 323
bitwise_xor() (odl.util.ufuncs.DiscreteLpUFuncs method), 335
bitwise_xor() (odl.util.ufuncs.NtuplesBaseUFuncs method), 348
bitwise_xor() (odl.util.ufuncs.NtuplesUFuncs method), 361
bitwise_xor() (odl.util.ufuncs.ProductSpaceUFuncs method), 374
broydens_first_method() (in module odl.solvers.findroot.newton), 195
broydens_second_method() (in module odl.solvers.findroot.newton), 196

C

CartesianProduct (class in odl.set.sets), 172
ceil() (odl.util.ufuncs.CudaNtuplesUFuncs method), 323
ceil() (odl.util.ufuncs.DiscreteLpUFuncs method), 336
ceil() (odl.util.ufuncs.NtuplesBaseUFuncs method), 348
ceil() (odl.util.ufuncs.NtuplesUFuncs method), 361
ceil() (odl.util.ufuncs.ProductSpaceUFuncs method), 374
cell_size (odl.discr.lp_discr.DiscreteLp attribute), 93
cell_size (odl.discr.lp_discr.DiscreteLpVector attribute), 97
cell_sizes() (odl.discr.grid.RegularGrid method), 78
cell_sizes() (odl.discr.grid.TensorGrid method), 86
cell_volume (odl.discr.grid.RegularGrid attribute), 74
cell_volume (odl.discr.lp_discr.DiscreteLp attribute), 93
cell_volume (odl.discr.lp_discr.DiscreteLpVector attribute), 97
center (odl.discr.grid.RegularGrid attribute), 74
Cn() (in module odl.space.ntuples), 297
coeff_size_list() (in module odl.trafos.wavelet), 310
collapse() (odl.set.domain.IntervalProd method), 160
ComplexNumbers (class in odl.set.sets), 174
ComponentProjection (class in odl.operator.pspace_ops), 148
ComponentProjectionAdjoint (class in odl.operator.pspace_ops), 150
conj() (odl.space.ntuples.FnVector method), 285
conj() (odl.util.ufuncs.CudaNtuplesUFuncs method), 323
conj() (odl.util.ufuncs.DiscreteLpUFuncs method), 336
conj() (odl.util.ufuncs.NtuplesBaseUFuncs method), 349
conj() (odl.util.ufuncs.NtuplesUFuncs method), 361
conj() (odl.util.ufuncs.ProductSpaceUFuncs method), 374
conjugate_gradient() (in module odl.solvers.iterative.iterative), 197
conjugate_gradient_normal() (in module odl.solvers.iterative.iterative), 197
const (odl.space.cu_ntuples.CudaFnConstWeighting attribute), 228
const (odl.space.cu_ntuples.CudaFnNoWeighting attribute), 233
const (odl.space.ntuples.FnConstWeighting attribute), 269
const (odl.space.ntuples.FnNoWeighting attribute), 277
ConstantLineSearch (class in odl.solvers.scalar.steplen), 203
ConstantOperator (class in odl.operator.default_ops), 102
contains() (odl.diagnostics.space.SpaceTest method), 35
contains_set() (odl.set.domain.IntervalProd method), 161
contains_set() (odl.set.sets.CartesianProduct method), 173
contains_set() (odl.set.sets.ComplexNumbers method), 174
contains_set() (odl.set.sets.EmptySet method), 175
contains_set() (odl.set.sets.Field method), 176
contains_set() (odl.set.sets.Integers method), 177
contains_set() (odl.set.sets.RealNumbers method), 178
contains_set() (odl.set.sets.Set method), 180
contains_set() (odl.set.sets.Strings method), 180
contains_set() (odl.set.sets.UniversalSet method), 181
convex_hull() (odl.discr.grid.RegularGrid method), 78
convex_hull() (odl.discr.grid.TensorGrid method), 86
coord_vectors (odl.discr.grid.RegularGrid attribute), 75
coord_vectors (odl.discr.grid.TensorGrid attribute), 83
copy() (odl.discr.discretization.DiscretizationVector method), 68
copy() (odl.discr.discretization.RawDiscretizationVector method), 73
copy() (odl.set.space.LinearSpaceVector method), 189
copy() (odl.space.base_ntuples.FnBaseVector method), 214
copy() (odl.space.base_ntuples.NtuplesBaseVector method), 220
copy() (odl.space.cu_ntuples.CudaNtuplesVector method), 249
copy() (odl.space.ntuples.NtuplesVector method), 296
copysign() (odl.util.ufuncs.CudaNtuplesUFuncs method), 323
copysign() (odl.util.ufuncs.DiscreteLpUFuncs method), 336
copysign() (odl.util.ufuncs.NtuplesBaseUFuncs method), 349
copysign() (odl.util.ufuncs.NtuplesUFuncs method), 362
copysign() (odl.util.ufuncs.ProductSpaceUFuncs method), 374
corners() (odl.discr.grid.RegularGrid method), 78
corners() (odl.discr.grid.TensorGrid method), 87
corners() (odl.set.domain.IntervalProd method), 161
cos() (odl.util.ufuncs.CudaNtuplesUFuncs method), 323

cos() (odl.util.ufuncs.DiscreteLpUFuncs method), 336
cos() (odl.util.ufuncs.NtuplesBaseUFuncs method), 349
cos() (odl.util.ufuncs.NtuplesUFuncs method), 362
cos() (odl.util.ufuncs.ProductSpaceUFuncs method), 374
cosh() (odl.util.ufuncs.CudaNtuplesUFuncs method), 323
cosh() (odl.util.ufuncs.DiscreteLpUFuncs method), 336
cosh() (odl.util.ufuncs.NtuplesBaseUFuncs method), 349
cosh() (odl.util.ufuncs.NtuplesUFuncs method), 362
cosh() (odl.util.ufuncs.ProductSpaceUFuncs method), 375
cu_weighted_dist() (in module odl.space.cu_ntuples), 250
cu_weighted_inner() (in module odl.space.cu_ntuples), 250
cu_weighted_norm() (in module odl.space.cu_ntuples), 251
Cuboid() (in module odl.set.domain), 164
CudaFn (class in odl.space.cu_ntuples), 220
CudaFnConstWeighting (class in odl.space.cu_ntuples), 227
CudaFnCustomDist (class in odl.space.cu_ntuples), 230
CudaFnCustomInnerProduct (class in odl.space.cu_ntuples), 231
CudaFnCustomNorm (class in odl.space.cu_ntuples), 232
CudaFnNoWeighting (class in odl.space.cu_ntuples), 233
CudaFnVector (class in odl.space.cu_ntuples), 235
CudaFnVectorWeighting (class in odl.space.cu_ntuples), 240
CudaNtuples (class in odl.space.cu_ntuples), 242
CudaNtuplesUFuncs (class in odl.util.ufuncs), 319
CudaNtuplesVector (class in odl.space.cu_ntuples), 244
CudaRn() (in module odl.space.cu_ntuples), 249

D

data (odl.space.cu_ntuples.CudaFnVector attribute), 236
data (odl.space.cu_ntuples.CudaNtuplesVector attribute), 244
data (odl.space.ntuples.FnVector attribute), 280
data (odl.space.ntuples.NtuplesVector attribute), 291
data_ptr (odl.space.cu_ntuples.CudaFnVector attribute), 236
data_ptr (odl.space.cu_ntuples.CudaNtuplesVector attribute), 245
data_ptr (odl.space.ntuples.FnVector attribute), 280
data_ptr (odl.space.ntuples.NtuplesVector attribute), 291
deg2rad() (odl.util.ufuncs.CudaNtuplesUFuncs method), 323
deg2rad() (odl.util.ufuncs.DiscreteLpUFuncs method), 336
deg2rad() (odl.util.ufuncs.NtuplesBaseUFuncs method), 349
deg2rad() (odl.util.ufuncs.NtuplesUFuncs method), 362
deg2rad() (odl.util.ufuncs.ProductSpaceUFuncs method), 375
derenzo_sources() (in module odl.util.phantom), 315
derivative() (odl.diagnostics.operator.OperatorTest method), 34
derivative() (odl.operator.default_ops.ResidualOperator method), 116
derivative() (odl.operator.operator.FunctionalLeftVectorMult method), 123
derivative() (odl.operator.operator.Operator method), 127
derivative() (odl.operator.operator.OperatorComp method), 129
derivative() (odl.operator.operator.OperatorLeftScalarMult method), 132
derivative() (odl.operator.operator.OperatorLeftVectorMult method), 135
derivative() (odl.operator.operator.OperatorPointwiseProduct method), 137
derivative() (odl.operator.operator.OperatorRightScalarMult method), 140
derivative() (odl.operator.operator.OperatorRightVectorMult method), 142
derivative() (odl.operator.operator.OperatorSum method), 145
DiscreteDivergence (class in odl.discr.discr_ops), 54
DiscreteGradient (class in odl.discr.discr_ops), 55
DiscreteLp (class in odl.discr.lp_discr), 92
DiscreteLpUFuncs (class in odl.util.ufuncs), 332
DiscreteLpVector (class in odl.discr.lp_discr), 96
DiscretePartDeriv (class in odl.discr.discr_ops), 58
DiscreteWaveletTransform (class in odl.trafos.wavelet), 306
DiscreteWaveletTransformInverse (class in odl.trafos.wavelet), 307
Discretization (class in odl.discr.discretization), 62
DiscretizationVector (class in odl.discr.discretization), 66
dist (odl.space.ntuples.FnCustomDist attribute), 271
dist() (odl.diagnostics.space.SpaceTest method), 35
dist() (odl.set.domain.IntervalProd method), 161
dist() (odl.set.space.LinearSpace method), 184
dist() (odl.set.space.LinearSpaceVector method), 189
dist() (odl.set.space.UniversalSpace method), 191
dist() (odl.space.base_ntuples.FnWeightingBase method), 215
dist() (odl.space.cu_ntuples.CudaFnConstWeighting method), 228
dist() (odl.space.cu_ntuples.CudaFnCustomDist method), 230
dist() (odl.space.cu_ntuples.CudaFnNoWeighting method), 234
dist() (odl.space.cu_ntuples.CudaFnVectorWeighting method), 241
dist() (odl.space.ntuples.FnConstWeighting method), 269
dist() (odl.space.ntuples.FnNoWeighting method), 278
divide() (odl.set.space.LinearSpace method), 184
divide() (odl.set.space.UniversalSpace method), 192

divide() (odl.util.ufuncs.CudaNtuplesUFuncs method), 324
divide() (odl.util.ufuncs.DiscreteLpUFuncs method), 336
divide() (odl.util.ufuncs.NtuplesBaseUFuncs method), 349
divide() (odl.util.ufuncs.NtuplesUFuncs method), 362
divide() (odl.util.ufuncs.ProductSpaceUFuncs method), 375
domain (odl.discr.discr_mappings.FunctionSetMapping attribute), 39
domain (odl.discr.discr_mappings.GridCollocation attribute), 41
domain (odl.discr.discr_mappings.LinearInterpolation attribute), 43
domain (odl.discr.discr_mappings.NearestInterpolation attribute), 45
domain (odl.discr.discr_mappings.RawGridCollocation attribute), 49
domain (odl.discr.discr_mappings.RawNearestInterpolation attribute), 51
domain (odl.discr.discr_ops.DiscreteDivergence attribute), 54
domain (odl.discr.discr_ops.DiscreteGradient attribute), 56
domain (odl.discr.discr_ops.DiscretePartDeriv attribute), 59
domain (odl.discr.discretization.Discretization attribute), 62
domain (odl.discr.discretization.RawDiscretization attribute), 69
domain (odl.discr.lp_discr.DiscreteLp attribute), 93
domain (odl.operator.default_ops.ConstantOperator attribute), 103
domain (odl.operator.default_ops.IdentityOperator attribute), 105
domain (odl.operator.default_ops.InnerProductAdjointOperator attribute), 107
domain (odl.operator.default_ops.InnerProductOperator attribute), 109
domain (odl.operator.default_ops.LinCombOperator attribute), 111
domain (odl.operator.default_ops.MultiplyOperator attribute), 113
domain (odl.operator.default_ops.ResidualOperator attribute), 114
domain (odl.operator.default_ops.ScalingOperator attribute), 116
domain (odl.operator.default_ops.ZeroOperator attribute), 119
domain (odl.operator.operator.FunctionalLeftVectorMult attribute), 122
domain (odl.operator.operator.Operator attribute), 125
domain (odl.operator.operator.OperatorComp attribute), 128
domain (odl.operator.operator.OperatorLeftScalarMult attribute), 130
domain (odl.operator.operator.OperatorLeftVectorMult attribute), 133
domain (odl.operator.operator.OperatorPointwiseProduct attribute), 135
domain (odl.operator.operator.OperatorRightScalarMult attribute), 138
domain (odl.operator.operator.OperatorRightVectorMult attribute), 141
domain (odl.operator.operator.OperatorSum attribute), 143
domain (odl.operator.pspace_ops.ComponentProjection attribute), 148
domain (odl.operator.pspace_ops.ComponentProjectionAdjoint attribute), 151
domain (odl.operator.pspace_ops.ProductSpaceOperator attribute), 155
domain (odl.space.fspace.FunctionSet attribute), 251
domain (odl.space.fspace.FunctionSetVector attribute), 253
domain (odl.space.fspace.FunctionSpace attribute), 255
domain (odl.space.fspace.FunctionSpaceVector attribute), 258
domain (odl.space.ntuples.MatVecOperator attribute), 288
domain (odl.tomo.backends.stir_bindings.BackProjectorByBinWrapper attribute), 301
domain (odl.tomo.backends.stir_bindings.ForwardProjectorByBinWrapper attribute), 302
domain (odl.trafos.wavelet.DiscreteWaveletTransform attribute), 306
domain (odl.trafos.wavelet.DiscreteWaveletTransformInverse attribute), 308
dspace (odl.discr.discretization.Discretization attribute), 62
dspace (odl.discr.discretization.RawDiscretization attribute), 69
dspace (odl.discr.lp_discr.DiscreteLp attribute), 93
dspace_type (odl.discr.discretization.Discretization attribute), 63
dspace_type (odl.discr.discretization.RawDiscretization attribute), 69
dspace_type (odl.discr.lp_discr.DiscreteLp attribute), 93
dspace_type() (in module odl.discr.discretization), 73
dtype (odl.discr.discretization.Discretization attribute), 63
dtype (odl.discr.discretization.DiscretizationVector attribute), 66
dtype (odl.discr.discretization.RawDiscretization attribute), 69
dtype (odl.discr.discretization.RawDiscretizationVector attribute), 71
dtype (odl.discr.lp_discr.DiscreteLp attribute), 93
dtype (odl.discr.lp_discr.DiscreteLpVector attribute), 97

dtype (odl.space.base_ntuples.FnBase attribute), 208
dtype (odl.space.base_ntuples.FnBaseVector attribute), 212
dtype (odl.space.base_ntuples.NtuplesBase attribute), 216
dtype (odl.space.base_ntuples.NtuplesBaseVector attribute), 218
dtype (odl.space.cu_ntuples.CudaFn attribute), 221
dtype (odl.space.cu_ntuples.CudaFnVector attribute), 236
dtype (odl.space.cu_ntuples.CudaNtuples attribute), 242
dtype (odl.space.cu_ntuples.CudaNtuplesVector attribute), 245
dtype (odl.space.ntuples.Fn attribute), 261
dtype (odl.space.ntuples.FnVector attribute), 280
dtype (odl.space.ntuples.Ntuples attribute), 289
dtype (odl.space.ntuples.NtuplesVector attribute), 292
dtype_repr() (in module odl.util.utility), 385

E

element() (odl.diagnostics.space.SpaceTest method), 36
element() (odl.discr.discretization.Discretization method), 65
element() (odl.discr.discretization.RawDiscretization method), 70
element() (odl.discr.grid.RegularGrid method), 79
element() (odl.discr.grid.TensorGrid method), 87
element() (odl.discr.lp_discr.DiscreteLp method), 95
element() (odl.set.domain.IntervalProd method), 162
element() (odl.set.pspace.ProductSpace method), 167
element() (odl.set.sets.CartesianProduct method), 174
element() (odl.set.sets.ComplexNumbers method), 175
element() (odl.set.sets.EmptySet method), 175
element() (odl.set.sets.Field method), 176
element() (odl.set.sets.Integers method), 177
element() (odl.set.sets.RealNumbers method), 178
element() (odl.set.sets.Set method), 180
element() (odl.set.sets.Strings method), 181
element() (odl.set.sets.UniversalSet method), 181
element() (odl.set.space.LinearSpace method), 184
element() (odl.set.space.UniversalSpace method), 192
element() (odl.space.cu_ntuples.CudaFn method), 225
element() (odl.space.cu_ntuples.CudaNtuples method), 243
element() (odl.space.fspace.FunctionSet method), 252
element() (odl.space.fspace.FunctionSpace method), 257
element() (odl.space.ntuples.Fn method), 266
element() (odl.space.ntuples.Ntuples method), 290
element_type (odl.discr.discretization.Discretization attribute), 63
element_type (odl.discr.discretization.RawDiscretization attribute), 69
element_type (odl.discr.lp_discr.DiscreteLp attribute), 93
element_type (odl.set.pspace.ProductSpace attribute), 165
element_type (odl.set.space.LinearSpace attribute), 182
element_type (odl.set.space.UniversalSpace attribute), 190
element_type (odl.space.base_ntuples.FnBase attribute), 209
element_type (odl.space.base_ntuples.NtuplesBase attribute), 216
element_type (odl.space.cu_ntuples.CudaFn attribute), 221
element_type (odl.space.cu_ntuples.CudaNtuples attribute), 242
element_type (odl.space.fspace.FunctionSet attribute), 251
element_type (odl.space.fspace.FunctionSpace attribute), 255
element_type (odl.space.ntuples.Fn attribute), 261
element_type (odl.space.ntuples.Ntuples attribute), 290
EmptySet (class in odl.set.sets), 175
end (odl.set.domain.IntervalProd attribute), 158
equal() (odl.util.ufuncs.CudaNtuplesUFuncs method), 324
equal() (odl.util.ufuncs.DiscreteLpUFuncs method), 337
equal() (odl.util.ufuncs.NtuplesBaseUFuncs method), 349
equal() (odl.util.ufuncs.NtuplesUFuncs method), 362
equal() (odl.util.ufuncs.ProductSpaceUFuncs method), 375
equals() (odl.diagnostics.space.SpaceTest method), 36
equiv() (odl.space.base_ntuples.FnWeightingBase method), 215
equiv() (odl.space.cu_ntuples.CudaFnConstWeighting method), 229
equiv() (odl.space.cu_ntuples.CudaFnNoWeighting method), 234
equiv() (odl.space.cu_ntuples.CudaFnVectorWeighting method), 241
equiv() (odl.space.ntuples.FnConstWeighting method), 270
equiv() (odl.space.ntuples.FnMatrixWeighting method), 275
equiv() (odl.space.ntuples.FnNoWeighting method), 278
equiv() (odl.space.ntuples.FnVectorWeighting method), 286
exp() (odl.util.ufuncs.CudaNtuplesUFuncs method), 324
exp() (odl.util.ufuncs.DiscreteLpUFuncs method), 337
exp() (odl.util.ufuncs.NtuplesBaseUFuncs method), 350
exp() (odl.util.ufuncs.NtuplesUFuncs method), 362
exp() (odl.util.ufuncs.ProductSpaceUFuncs method), 375
exp2() (odl.util.ufuncs.CudaNtuplesUFuncs method), 324
exp2() (odl.util.ufuncs.DiscreteLpUFuncs method), 337
exp2() (odl.util.ufuncs.NtuplesBaseUFuncs method), 350
exp2() (odl.util.ufuncs.NtuplesUFuncs method), 363
exp2() (odl.util.ufuncs.ProductSpaceUFuncs method), 375

exp_zero_seq() (in module odl.solvers.iterative.iterative), 198
expm1() (odl.util.ufuncs.CudaNtuplesUFuncs method), 324
expm1() (odl.util.ufuncs.DiscreteLpUFuncs method), 337
expm1() (odl.util.ufuncs.NtuplesBaseUFuncs method), 350
expm1() (odl.util.ufuncs.NtuplesUFuncs method), 363
expm1() (odl.util.ufuncs.ProductSpaceUFuncs method), 375
exponent (odl.discr.lp_discr.DiscreteLp attribute), 93
exponent (odl.space.base_ntuples.FnWeightingBase attribute), 214
exponent (odl.space.cu_ntuples.CudaFn attribute), 221
exponent (odl.space.cu_ntuples.CudaFnConstWeighting attribute), 228
exponent (odl.space.cu_ntuples.CudaFnCustomDist attribute), 230
exponent (odl.space.cu_ntuples.CudaFnCustomInnerProduct attribute), 231
exponent (odl.space.cu_ntuples.CudaFnCustomNorm attribute), 232
exponent (odl.space.cu_ntuples.CudaFnNoWeighting attribute), 234
exponent (odl.space.cu_ntuples.CudaFnVectorWeighting attribute), 240
exponent (odl.space.ntuples.Fn attribute), 261
exponent (odl.space.ntuples.FnConstWeighting attribute), 269
exponent (odl.space.ntuples.FnCustomDist attribute), 271
exponent (odl.space.ntuples.FnCustomInnerProduct attribute), 272
exponent (odl.space.ntuples.FnCustomNorm attribute), 273
exponent (odl.space.ntuples.FnMatrixWeighting attribute), 275
exponent (odl.space.ntuples.FnNoWeighting attribute), 277
exponent (odl.space.ntuples.FnVectorWeighting attribute), 286
extension (odl.discr.discretization.Discretization attribute), 63
extension (odl.discr.discretization.RawDiscretization attribute), 69
extension (odl.discr.lp_discr.DiscreteLp attribute), 94

F

fail() (odl.util.testutils.FailCounter method), 316
FailCounter (class in odl.util.testutils), 316
Field (class in odl.set.sets), 176
field (odl.discr.discretization.Discretization attribute), 63
field (odl.discr.lp_discr.DiscreteLp attribute), 94
field (odl.set.pspace.ProductSpace attribute), 165
field (odl.set.sets.ComplexNumbers attribute), 174
field (odl.set.sets.Field attribute), 176
field (odl.set.sets.RealNumbers attribute), 178
field (odl.set.space.LinearSpace attribute), 182
field (odl.set.space.UniversalSpace attribute), 190
field (odl.space.base_ntuples.FnBase attribute), 209
field (odl.space.cu_ntuples.CudaFn attribute), 221
field (odl.space.fspace.FunctionSpace attribute), 255
field (odl.space.ntuples.Fn attribute), 261
field() (odl.diagnostics.space.SpaceTest method), 36
finite_diff() (in module odl.discr.discr_ops), 61
floor() (odl.util.ufuncs.CudaNtuplesUFuncs method), 324
floor() (odl.util.ufuncs.DiscreteLpUFuncs method), 337
floor() (odl.util.ufuncs.NtuplesBaseUFuncs method), 350
floor() (odl.util.ufuncs.NtuplesUFuncs method), 363
floor() (odl.util.ufuncs.ProductSpaceUFuncs method), 376
floor_divide() (odl.util.ufuncs.CudaNtuplesUFuncs method), 324
floor_divide() (odl.util.ufuncs.DiscreteLpUFuncs method), 337
floor_divide() (odl.util.ufuncs.NtuplesBaseUFuncs method), 350
floor_divide() (odl.util.ufuncs.NtuplesUFuncs method), 363
floor_divide() (odl.util.ufuncs.ProductSpaceUFuncs method), 376
fmax() (odl.util.ufuncs.CudaNtuplesUFuncs method), 325
fmax() (odl.util.ufuncs.DiscreteLpUFuncs method), 337
fmax() (odl.util.ufuncs.NtuplesBaseUFuncs method), 350
fmax() (odl.util.ufuncs.NtuplesUFuncs method), 363
fmax() (odl.util.ufuncs.ProductSpaceUFuncs method), 376
fmin() (odl.util.ufuncs.CudaNtuplesUFuncs method), 325
fmin() (odl.util.ufuncs.DiscreteLpUFuncs method), 338
fmin() (odl.util.ufuncs.NtuplesBaseUFuncs method), 350
fmin() (odl.util.ufuncs.NtuplesUFuncs method), 363
fmin() (odl.util.ufuncs.ProductSpaceUFuncs method), 376
fmod() (odl.util.ufuncs.CudaNtuplesUFuncs method), 325
fmod() (odl.util.ufuncs.DiscreteLpUFuncs method), 338
fmod() (odl.util.ufuncs.NtuplesBaseUFuncs method), 351
fmod() (odl.util.ufuncs.NtuplesUFuncs method), 363
fmod() (odl.util.ufuncs.ProductSpaceUFuncs method), 376
Fn (class in odl.space.ntuples), 260
FnBase (class in odl.space.base_ntuples), 208
FnBaseVector (class in odl.space.base_ntuples), 211
FnConstWeighting (class in odl.space.ntuples), 268
FnCustomDist (class in odl.space.ntuples), 271
FnCustomInnerProduct (class in odl.space.ntuples), 272
FnCustomNorm (class in odl.space.ntuples), 273
FnMatrixWeighting (class in odl.space.ntuples), 274

FnNoWeighting (class in odl.space.ntuples), 277
FnVector (class in odl.space.ntuples), 279
FnVectorWeighting (class in odl.space.ntuples), 285
FnWeightingBase (class in odl.space.base_ntuples), 214
ForEachPartial (class in odl.solvers.util.partial), 205
ForwardProjectorByBinWrapper (class in odl.tomo.backends.stir_bindings), 302
FunctionalLeftVectorMult (class in odl.operator.operator), 121
FunctionSet (class in odl.space.fspace), 251
FunctionSetMapping (class in odl.discr.discr_mappings), 38
FunctionSetVector (class in odl.space.fspace), 253
FunctionSpace (class in odl.space.fspace), 255
FunctionSpaceVector (class in odl.space.fspace), 257

G

gauss_newton() (in module odl.solvers.iterative.iterative), 198
greater() (odl.util.ufuncs.CudaNtuplesUFuncs method), 325
greater() (odl.util.ufuncs.DiscreteLpUFuncs method), 338
greater() (odl.util.ufuncs.NtuplesBaseUFuncs method), 351
greater() (odl.util.ufuncs.NtuplesUFuncs method), 364
greater() (odl.util.ufuncs.ProductSpaceUFuncs method), 376
greater_equal() (odl.util.ufuncs.CudaNtuplesUFuncs method), 325
greater_equal() (odl.util.ufuncs.DiscreteLpUFuncs method), 338
greater_equal() (odl.util.ufuncs.NtuplesBaseUFuncs method), 351
greater_equal() (odl.util.ufuncs.NtuplesUFuncs method), 364
greater_equal() (odl.util.ufuncs.ProductSpaceUFuncs method), 376
grid (odl.discr.discr_mappings.FunctionSetMapping attribute), 39
grid (odl.discr.discr_mappings.GridCollocation attribute), 41
grid (odl.discr.discr_mappings.LinearInterpolation attribute), 43
grid (odl.discr.discr_mappings.NearestInterpolation attribute), 45
grid (odl.discr.discr_mappings.RawGridCollocation attribute), 49
grid (odl.discr.discr_mappings.RawNearestInterpolation attribute), 51
grid (odl.discr.lp_discr.DiscreteLp attribute), 94
GridCollocation (class in odl.discr.discr_mappings), 40

H

hypot() (odl.util.ufuncs.CudaNtuplesUFuncs method), 325
hypot() (odl.util.ufuncs.DiscreteLpUFuncs method), 338
hypot() (odl.util.ufuncs.NtuplesBaseUFuncs method), 351
hypot() (odl.util.ufuncs.NtuplesUFuncs method), 364
hypot() (odl.util.ufuncs.ProductSpaceUFuncs method), 377

I

IdentityOperator (class in odl.operator.default_ops), 104
imag (odl.space.ntuples.FnVector attribute), 280
impl (odl.space.base_ntuples.FnWeightingBase attribute), 214
impl (odl.space.cu_ntuples.CudaFnConstWeighting attribute), 228
impl (odl.space.cu_ntuples.CudaFnCustomDist attribute), 230
impl (odl.space.cu_ntuples.CudaFnCustomInnerProduct attribute), 231
impl (odl.space.cu_ntuples.CudaFnCustomNorm attribute), 232
impl (odl.space.cu_ntuples.CudaFnNoWeighting attribute), 234
impl (odl.space.cu_ntuples.CudaFnVectorWeighting attribute), 240
impl (odl.space.ntuples.FnConstWeighting attribute), 269
impl (odl.space.ntuples.FnCustomDist attribute), 271
impl (odl.space.ntuples.FnCustomInnerProduct attribute), 272
impl (odl.space.ntuples.FnCustomNorm attribute), 273
impl (odl.space.ntuples.FnMatrixWeighting attribute), 275
impl (odl.space.ntuples.FnNoWeighting attribute), 277
impl (odl.space.ntuples.FnVectorWeighting attribute), 286
index (odl.operator.pspace_ops.ComponentProjection attribute), 148
index (odl.operator.pspace_ops.ComponentProjectionAdjoint attribute), 151
inner (odl.space.ntuples.FnCustomInnerProduct attribute), 272
inner() (odl.diagnostics.space.SpaceTest method), 36
inner() (odl.set.space.LinearSpace method), 185
inner() (odl.set.space.LinearSpaceVector method), 189
inner() (odl.set.space.UniversalSpace method), 192
inner() (odl.space.base_ntuples.FnWeightingBase method), 215
inner() (odl.space.cu_ntuples.CudaFnConstWeighting method), 229
inner() (odl.space.cu_ntuples.CudaFnCustomDist method), 230

inner() (odl.space.cu_ntuples.CudaFnCustomInnerProduct method), 231
inner() (odl.space.cu_ntuples.CudaFnCustomNorm method), 233
inner() (odl.space.cu_ntuples.CudaFnNoWeighting method), 234
inner() (odl.space.cu_ntuples.CudaFnVectorWeighting method), 241
inner() (odl.space.ntuples.FnConstWeighting method), 270
inner() (odl.space.ntuples.FnCustomDist method), 271
inner() (odl.space.ntuples.FnCustomNorm method), 274
inner() (odl.space.ntuples.FnMatrixWeighting method), 276
inner() (odl.space.ntuples.FnNoWeighting method), 278
inner() (odl.space.ntuples.FnVectorWeighting method), 287
InnerProductAdjointOperator (class in odl.operator.default_ops), 107
InnerProductOperator (class in odl.operator.default_ops), 109
insert() (odl.discr.grid.RegularGrid method), 79
insert() (odl.discr.grid.TensorGrid method), 87
insert() (odl.set.domain.IntervalProd method), 162
Integers (class in odl.set.sets), 177
interp (odl.discr.lp_discr.DiscreteLp attribute), 94
Interval() (in module odl.set.domain), 164
IntervalProd (class in odl.set.domain), 157
inverse (odl.discr.discr_mappings.FunctionSetMapping attribute), 39
inverse (odl.discr.discr_mappings.GridCollocation attribute), 41
inverse (odl.discr.discr_mappings.LinearInterpolation attribute), 44
inverse (odl.discr.discr_mappings.NearestInterpolation attribute), 46
inverse (odl.discr.discr_mappings.RawGridCollocation attribute), 49
inverse (odl.discr.discr_mappings.RawNearestInterpolation attribute), 51
inverse (odl.discr.discr_ops.DiscreteDivergence attribute), 54
inverse (odl.discr.discr_ops.DiscreteGradient attribute), 56
inverse (odl.discr.discr_ops.DiscretePartDeriv attribute), 59
inverse (odl.operator.default_ops.ConstantOperator attribute), 103
inverse (odl.operator.default_ops.IdentityOperator attribute), 105
inverse (odl.operator.default_ops.InnerProductAdjointOperator attribute), 107
inverse (odl.operator.default_ops.InnerProductOperator attribute), 109
inverse (odl.operator.default_ops.LinCombOperator attribute), 111
inverse (odl.operator.default_ops.MultiplyOperator attribute), 113
inverse (odl.operator.default_ops.ResidualOperator attribute), 114
inverse (odl.operator.default_ops.ScalingOperator attribute), 117
inverse (odl.operator.default_ops.ZeroOperator attribute), 119
inverse (odl.operator.operator.FunctionalLeftVectorMult attribute), 122
inverse (odl.operator.operator.Operator attribute), 126
inverse (odl.operator.operator.OperatorComp attribute), 128
inverse (odl.operator.operator.OperatorLeftScalarMult attribute), 130
inverse (odl.operator.operator.OperatorLeftVectorMult attribute), 133
inverse (odl.operator.operator.OperatorPointwiseProduct attribute), 135
inverse (odl.operator.operator.OperatorRightScalarMult attribute), 138
inverse (odl.operator.operator.OperatorRightVectorMult attribute), 141
inverse (odl.operator.operator.OperatorSum attribute), 143
inverse (odl.operator.pspace_ops.ComponentProjection attribute), 149
inverse (odl.operator.pspace_ops.ComponentProjectionAdjoint attribute), 151
inverse (odl.operator.pspace_ops.ProductSpaceOperator attribute), 155
inverse (odl.space.fspace.FunctionSetVector attribute), 253
inverse (odl.space.fspace.FunctionSpaceVector attribute), 258
inverse (odl.space.ntuples.MatVecOperator attribute), 288
inverse (odl.tomo.backends.stir_bindings.BackProjectorByBinWrapper attribute), 301
inverse (odl.tomo.backends.stir_bindings.ForwardProjectorByBinWrapper attribute), 303
inverse (odl.trafos.wavelet.DiscreteWaveletTransform attribute), 306
inverse (odl.trafos.wavelet.DiscreteWaveletTransformInverse attribute), 308
invert() (odl.util.ufuncs.CudaNtuplesUFuncs method), 325
invert() (odl.util.ufuncs.DiscreteLpUFuncs method), 338
invert() (odl.util.ufuncs.NtuplesBaseUFuncs method), 351
invert() (odl.util.ufuncs.NtuplesUFuncs method), 364
invert() (odl.util.ufuncs.ProductSpaceUFuncs method), 377

is_biorthogonal (odl.trafos.wavelet.DiscreteWaveletTransform attribute), 306
is_biorthogonal (odl.trafos.wavelet.DiscreteWaveletTransformInverse attribute), 308
is_cn (odl.discr.discretization.Discretization attribute), 63
is_cn (odl.discr.lp_discr.DiscreteLp attribute), 94
is_cn (odl.space.base_ntuples.FnBase attribute), 209
is_cn (odl.space.cu_ntuples.CudaFn attribute), 221
is_cn (odl.space.ntuples.Fn attribute), 261
is_complex_floating_dtype() (in module odl.util.utility), 385
is_floating_dtype() (in module odl.util.utility), 385
is_functional (odl.discr.discr_mappings.FunctionSetMapping attribute), 39
is_functional (odl.discr.discr_mappings.GridCollocation attribute), 41
is_functional (odl.discr.discr_mappings.LinearInterpolation attribute), 44
is_functional (odl.discr.discr_mappings.NearestInterpolation attribute), 46
is_functional (odl.discr.discr_mappings.RawGridCollocation attribute), 49
is_functional (odl.discr.discr_mappings.RawNearestInterpolation attribute), 52
is_functional (odl.discr.discr_ops.DiscreteDivergence attribute), 54
is_functional (odl.discr.discr_ops.DiscreteGradient attribute), 56
is_functional (odl.discr.discr_ops.DiscretePartDeriv attribute), 59
is_functional (odl.operator.default_ops.ConstantOperator attribute), 103
is_functional (odl.operator.default_ops.IdentityOperator attribute), 105
is_functional (odl.operator.default_ops.InnerProductAdjointOperator attribute), 107
is_functional (odl.operator.default_ops.InnerProductOperator attribute), 109
is_functional (odl.operator.default_ops.LinCombOperator attribute), 111
is_functional (odl.operator.default_ops.MultiplyOperator attribute), 113
is_functional (odl.operator.default_ops.ResidualOperator attribute), 114
is_functional (odl.operator.default_ops.ScalingOperator attribute), 117
is_functional (odl.operator.default_ops.ZeroOperator attribute), 119
is_functional (odl.operator.operator.FunctionalLeftVectorMult attribute), 122
is_functional (odl.operator.operator.Operator attribute), 126
is_functional (odl.operator.operator.OperatorComp attribute), 128
is_functional (odl.operator.operator.OperatorLeftScalarMult attribute), 131
is_functional (odl.operator.operator.OperatorLeftVectorMult attribute), 133
is_functional (odl.operator.operator.OperatorPointwiseProduct attribute), 136
is_functional (odl.operator.operator.OperatorRightScalarMult attribute), 138
is_functional (odl.operator.operator.OperatorRightVectorMult attribute), 141
is_functional (odl.operator.operator.OperatorSum attribute), 143
is_functional (odl.operator.pspace_ops.ComponentProjection attribute), 149
is_functional (odl.operator.pspace_ops.ComponentProjectionAdjoint attribute), 151
is_functional (odl.operator.pspace_ops.ProductSpaceOperator attribute), 155
is_functional (odl.space.fspace.FunctionSetVector attribute), 253
is_functional (odl.space.fspace.FunctionSpaceVector attribute), 259
is_functional (odl.space.ntuples.MatVecOperator attribute), 288
is_functional (odl.tomo.backends.stir_bindings.BackProjectorByBinWrappe attribute), 301
is_functional (odl.tomo.backends.stir_bindings.ForwardProjectorByBinWra attribute), 303
is_functional (odl.trafos.wavelet.DiscreteWaveletTransform attribute), 306
is_functional (odl.trafos.wavelet.DiscreteWaveletTransformInverse attribute), 308
is_linear (odl.discr.discr_mappings.FunctionSetMapping attribute), 39
is_linear (odl.discr.discr_mappings.GridCollocation attribute), 41
is_linear (odl.discr.discr_mappings.LinearInterpolation attribute), 44
is_linear (odl.discr.discr_mappings.NearestInterpolation attribute), 46
is_linear (odl.discr.discr_mappings.RawGridCollocation attribute), 49
is_linear (odl.discr.discr_mappings.RawNearestInterpolation attribute), 52
is_linear (odl.discr.discr_ops.DiscreteDivergence attribute), 54
is_linear (odl.discr.discr_ops.DiscreteGradient attribute), 56
is_linear (odl.discr.discr_ops.DiscretePartDeriv attribute), 59
is_linear (odl.operator.default_ops.ConstantOperator attribute), 103
is_linear (odl.operator.default_ops.IdentityOperator attribute), 105

is_linear (odl.operator.default_ops.InnerProductAdjointOperator attribute), 107
is_linear (odl.operator.default_ops.InnerProductOperator attribute), 109
is_linear (odl.operator.default_ops.LinCombOperator attribute), 111
is_linear (odl.operator.default_ops.MultiplyOperator attribute), 113
is_linear (odl.operator.default_ops.ResidualOperator attribute), 115
is_linear (odl.operator.default_ops.ScalingOperator attribute), 117
is_linear (odl.operator.default_ops.ZeroOperator attribute), 119
is_linear (odl.operator.operator.FunctionalLeftVectorMult attribute), 122
is_linear (odl.operator.operator.Operator attribute), 126
is_linear (odl.operator.operator.OperatorComp attribute), 128
is_linear (odl.operator.operator.OperatorLeftScalarMult attribute), 131
is_linear (odl.operator.operator.OperatorLeftVectorMult attribute), 133
is_linear (odl.operator.operator.OperatorPointwiseProduct attribute), 136
is_linear (odl.operator.operator.OperatorRightScalarMult attribute), 138
is_linear (odl.operator.operator.OperatorRightVectorMult attribute), 141
is_linear (odl.operator.operator.OperatorSum attribute), 143
is_linear (odl.operator.pspace_ops.ComponentProjection attribute), 149
is_linear (odl.operator.pspace_ops.ComponentProjectionAdjoint attribute), 151
is_linear (odl.operator.pspace_ops.ProductSpaceOperator attribute), 155
is_linear (odl.space.fspace.FunctionSetVector attribute), 254
is_linear (odl.space.fspace.FunctionSpaceVector attribute), 259
is_linear (odl.space.ntuples.MatVecOperator attribute), 288
is_linear (odl.tomo.backends.stir_bindings.BackProjectorByBinWrapper attribute), 301
is_linear (odl.tomo.backends.stir_bindings.ForwardProjectorByBinWrapper attribute), 303
is_linear (odl.trafos.wavelet.DiscreteWaveletTransform attribute), 306
is_linear (odl.trafos.wavelet.DiscreteWaveletTransformInverse attribute), 308
is_orthogonal (odl.trafos.wavelet.DiscreteWaveletTransform attribute), 307
is_orthogonal (odl.trafos.wavelet.DiscreteWaveletTransformInverse attribute), 308
is_real_dtype() (in module odl.util.utility), 385
is_real_floating_dtype() (in module odl.util.utility), 386
is_rn (odl.discr.discretization.Discretization attribute), 63
is_rn (odl.discr.lp_discr.DiscreteLp attribute), 94
is_rn (odl.space.base_ntuples.FnBase attribute), 209
is_rn (odl.space.cu_ntuples.CudaFn attribute), 221
is_rn (odl.space.ntuples.Fn attribute), 261
is_scalar_dtype() (in module odl.util.utility), 386
is_subdict() (in module odl.util.testutils), 318
is_subgrid() (odl.discr.grid.RegularGrid method), 79
is_subgrid() (odl.discr.grid.TensorGrid method), 88
isfinite() (odl.util.ufuncs.CudaNtuplesUFuncs method), 326
isfinite() (odl.util.ufuncs.DiscreteLpUFuncs method), 338
isfinite() (odl.util.ufuncs.NtuplesBaseUFuncs method), 351
isfinite() (odl.util.ufuncs.NtuplesUFuncs method), 364
isfinite() (odl.util.ufuncs.ProductSpaceUFuncs method), 377
isinf() (odl.util.ufuncs.CudaNtuplesUFuncs method), 326
isinf() (odl.util.ufuncs.DiscreteLpUFuncs method), 339
isinf() (odl.util.ufuncs.NtuplesBaseUFuncs method), 351
isinf() (odl.util.ufuncs.NtuplesUFuncs method), 364
isinf() (odl.util.ufuncs.ProductSpaceUFuncs method), 377
isnan() (odl.util.ufuncs.CudaNtuplesUFuncs method), 326
isnan() (odl.util.ufuncs.DiscreteLpUFuncs method), 339
isnan() (odl.util.ufuncs.NtuplesBaseUFuncs method), 352
isnan() (odl.util.ufuncs.NtuplesUFuncs method), 364
isnan() (odl.util.ufuncs.ProductSpaceUFuncs method), 377
itemsize (odl.discr.discretization.DiscretizationVector attribute), 66
itemsize (odl.discr.discretization.RawDiscretizationVector attribute), 71
itemsize (odl.discr.lp_discr.DiscreteLpVector attribute), 97
itemsize (odl.space.base_ntuples.FnBaseVector attribute), 212
itemsize (odl.space.base_ntuples.NtuplesBaseVector attribute), 218
itemsize (odl.space.cu_ntuples.CudaFnVector attribute), 236
itemsize (odl.space.cu_ntuples.CudaNtuplesVector attribute), 245
itemsize (odl.space.ntuples.FnVector attribute), 281
itemsize (odl.space.ntuples.NtuplesVector attribute), 292

L

landweber() (in module odl.solvers.iterative.iterative), 199

left_shift() (odl.util.ufuncs.CudaNtuplesUFuncs method), 326
left_shift() (odl.util.ufuncs.DiscreteLpUFuncs method), 339
left_shift() (odl.util.ufuncs.NtuplesBaseUFuncs method), 352
left_shift() (odl.util.ufuncs.NtuplesUFuncs method), 365
left_shift() (odl.util.ufuncs.ProductSpaceUFuncs method), 377
length (odl.set.domain.IntervalProd attribute), 158
length (odl.set.sets.Strings attribute), 180
less() (odl.util.ufuncs.CudaNtuplesUFuncs method), 326
less() (odl.util.ufuncs.DiscreteLpUFuncs method), 339
less() (odl.util.ufuncs.NtuplesBaseUFuncs method), 352
less() (odl.util.ufuncs.NtuplesUFuncs method), 365
less() (odl.util.ufuncs.ProductSpaceUFuncs method), 377
less_equal() (odl.util.ufuncs.CudaNtuplesUFuncs method), 326
less_equal() (odl.util.ufuncs.DiscreteLpUFuncs method), 339
less_equal() (odl.util.ufuncs.NtuplesBaseUFuncs method), 352
less_equal() (odl.util.ufuncs.NtuplesUFuncs method), 365
less_equal() (odl.util.ufuncs.ProductSpaceUFuncs method), 378
lincomb() (odl.set.space.LinearSpace method), 185
lincomb() (odl.set.space.LinearSpaceVector method), 189
lincomb() (odl.set.space.UniversalSpace method), 192
LinCombOperator (class in odl.operator.default_ops), 110
linear() (odl.diagnostics.operator.OperatorTest method), 34
LinearInterpolation (class in odl.discr.discr_mappings), 43
linearity() (odl.diagnostics.space.SpaceTest method), 36
LinearSpace (class in odl.set.space), 182
LinearSpaceNotImplementedError, 187
LinearSpaceTypeError, 187
LinearSpaceVector (class in odl.set.space), 187
LineSearch (class in odl.solvers.scalar.steplen), 203
log() (odl.util.ufuncs.CudaNtuplesUFuncs method), 326
log() (odl.util.ufuncs.DiscreteLpUFuncs method), 339
log() (odl.util.ufuncs.NtuplesBaseUFuncs method), 352
log() (odl.util.ufuncs.NtuplesUFuncs method), 365
log() (odl.util.ufuncs.ProductSpaceUFuncs method), 378
log10() (odl.util.ufuncs.CudaNtuplesUFuncs method), 327
log10() (odl.util.ufuncs.DiscreteLpUFuncs method), 339
log10() (odl.util.ufuncs.NtuplesBaseUFuncs method), 352
log10() (odl.util.ufuncs.NtuplesUFuncs method), 365
log10() (odl.util.ufuncs.ProductSpaceUFuncs method), 378
log1p() (odl.util.ufuncs.CudaNtuplesUFuncs method), 327
log1p() (odl.util.ufuncs.DiscreteLpUFuncs method), 340
log1p() (odl.util.ufuncs.NtuplesBaseUFuncs method), 352
log1p() (odl.util.ufuncs.NtuplesUFuncs method), 365
log1p() (odl.util.ufuncs.ProductSpaceUFuncs method), 378
log2() (odl.util.ufuncs.CudaNtuplesUFuncs method), 327
log2() (odl.util.ufuncs.DiscreteLpUFuncs method), 340
log2() (odl.util.ufuncs.NtuplesBaseUFuncs method), 353
log2() (odl.util.ufuncs.NtuplesUFuncs method), 365
log2() (odl.util.ufuncs.ProductSpaceUFuncs method), 378
logaddexp() (odl.util.ufuncs.CudaNtuplesUFuncs method), 327
logaddexp() (odl.util.ufuncs.DiscreteLpUFuncs method), 340
logaddexp() (odl.util.ufuncs.NtuplesBaseUFuncs method), 353
logaddexp() (odl.util.ufuncs.NtuplesUFuncs method), 366
logaddexp() (odl.util.ufuncs.ProductSpaceUFuncs method), 378
logaddexp2() (odl.util.ufuncs.CudaNtuplesUFuncs method), 327
logaddexp2() (odl.util.ufuncs.DiscreteLpUFuncs method), 340
logaddexp2() (odl.util.ufuncs.NtuplesBaseUFuncs method), 353
logaddexp2() (odl.util.ufuncs.NtuplesUFuncs method), 366
logaddexp2() (odl.util.ufuncs.ProductSpaceUFuncs method), 378
logical_and() (odl.util.ufuncs.CudaNtuplesUFuncs method), 327
logical_and() (odl.util.ufuncs.DiscreteLpUFuncs method), 340
logical_and() (odl.util.ufuncs.NtuplesBaseUFuncs method), 353
logical_and() (odl.util.ufuncs.NtuplesUFuncs method), 366
logical_and() (odl.util.ufuncs.ProductSpaceUFuncs method), 379
logical_not() (odl.util.ufuncs.CudaNtuplesUFuncs method), 327
logical_not() (odl.util.ufuncs.DiscreteLpUFuncs method), 340
logical_not() (odl.util.ufuncs.NtuplesBaseUFuncs method), 353
logical_not() (odl.util.ufuncs.NtuplesUFuncs method), 366
logical_not() (odl.util.ufuncs.ProductSpaceUFuncs method), 379

logical_or() (odl.util.ufuncs.CudaNtuplesUFuncs method), 328
logical_or() (odl.util.ufuncs.DiscreteLpUFuncs method), 340
logical_or() (odl.util.ufuncs.NtuplesBaseUFuncs method), 353
logical_or() (odl.util.ufuncs.NtuplesUFuncs method), 366
logical_or() (odl.util.ufuncs.ProductSpaceUFuncs method), 379
logical_xor() (odl.util.ufuncs.CudaNtuplesUFuncs method), 328
logical_xor() (odl.util.ufuncs.DiscreteLpUFuncs method), 341
logical_xor() (odl.util.ufuncs.NtuplesBaseUFuncs method), 353
logical_xor() (odl.util.ufuncs.NtuplesUFuncs method), 366
logical_xor() (odl.util.ufuncs.ProductSpaceUFuncs method), 379

M

matrix (odl.space.ntuples.FnMatrixWeighting attribute), 275
matrix (odl.space.ntuples.MatVecOperator attribute), 288
matrix_issparse (odl.space.ntuples.FnMatrixWeighting attribute), 275
matrix_issparse (odl.space.ntuples.MatVecOperator attribute), 288
matrix_isvalid() (odl.space.ntuples.FnMatrixWeighting method), 276
matrix_representation() (in module odl.operator.oputils), 147
MatVecOperator (class in odl.space.ntuples), 288
max() (odl.discr.grid.RegularGrid method), 80
max() (odl.discr.grid.TensorGrid method), 88
max() (odl.set.domain.IntervalProd method), 163
max() (odl.util.ufuncs.CudaNtuplesUFuncs method), 328
max() (odl.util.ufuncs.DiscreteLpUFuncs method), 341
max() (odl.util.ufuncs.NtuplesBaseUFuncs method), 354
max() (odl.util.ufuncs.NtuplesUFuncs method), 366
max() (odl.util.ufuncs.ProductSpaceUFuncs method), 379
max_pt (odl.discr.grid.RegularGrid attribute), 75
max_pt (odl.discr.grid.TensorGrid attribute), 83
maximum() (odl.util.ufuncs.CudaNtuplesUFuncs method), 328
maximum() (odl.util.ufuncs.DiscreteLpUFuncs method), 341
maximum() (odl.util.ufuncs.NtuplesBaseUFuncs method), 354
maximum() (odl.util.ufuncs.NtuplesUFuncs method), 367
maximum() (odl.util.ufuncs.ProductSpaceUFuncs method), 379
measure() (odl.set.domain.IntervalProd method), 163
meshgrid() (odl.discr.grid.RegularGrid method), 80
meshgrid() (odl.discr.grid.TensorGrid method), 89
method() (in module odl.util.ufuncs), 383
midpoint (odl.set.domain.IntervalProd attribute), 158
min() (odl.discr.grid.RegularGrid method), 81
min() (odl.discr.grid.TensorGrid method), 89
min() (odl.set.domain.IntervalProd method), 163
min() (odl.util.ufuncs.CudaNtuplesUFuncs method), 328
min() (odl.util.ufuncs.DiscreteLpUFuncs method), 341
min() (odl.util.ufuncs.NtuplesBaseUFuncs method), 354
min() (odl.util.ufuncs.NtuplesUFuncs method), 367
min() (odl.util.ufuncs.ProductSpaceUFuncs method), 379
min_pt (odl.discr.grid.RegularGrid attribute), 75
min_pt (odl.discr.grid.TensorGrid attribute), 83
minimum() (odl.util.ufuncs.CudaNtuplesUFuncs method), 328
minimum() (odl.util.ufuncs.DiscreteLpUFuncs method), 341
minimum() (odl.util.ufuncs.NtuplesBaseUFuncs method), 354
minimum() (odl.util.ufuncs.NtuplesUFuncs method), 367
minimum() (odl.util.ufuncs.ProductSpaceUFuncs method), 380
mod() (odl.util.ufuncs.CudaNtuplesUFuncs method), 328
mod() (odl.util.ufuncs.DiscreteLpUFuncs method), 341
mod() (odl.util.ufuncs.NtuplesBaseUFuncs method), 354
mod() (odl.util.ufuncs.NtuplesUFuncs method), 367
mod() (odl.util.ufuncs.ProductSpaceUFuncs method), 380
modf() (odl.util.ufuncs.CudaNtuplesUFuncs method), 329
modf() (odl.util.ufuncs.DiscreteLpUFuncs method), 341
modf() (odl.util.ufuncs.NtuplesBaseUFuncs method), 354
modf() (odl.util.ufuncs.NtuplesUFuncs method), 367
modf() (odl.util.ufuncs.ProductSpaceUFuncs method), 380
multiply() (odl.diagnostics.space.SpaceTest method), 37
multiply() (odl.set.space.LinearSpace method), 186
multiply() (odl.set.space.LinearSpaceVector method), 189
multiply() (odl.set.space.UniversalSpace method), 193
multiply() (odl.util.ufuncs.CudaNtuplesUFuncs method), 329
multiply() (odl.util.ufuncs.DiscreteLpUFuncs method), 342
multiply() (odl.util.ufuncs.NtuplesBaseUFuncs method), 354
multiply() (odl.util.ufuncs.NtuplesUFuncs method), 367
multiply() (odl.util.ufuncs.ProductSpaceUFuncs method), 380
MultiplyOperator (class in odl.operator.default_ops), 112

N

nbytes (odl.discr.discretization.DiscretizationVector attribute), 67
nbytes (odl.discr.discretization.RawDiscretizationVector attribute), 71
nbytes (odl.discr.lp_discr.DiscreteLpVector attribute), 97
nbytes (odl.space.base_ntuples.FnBaseVector attribute), 212
nbytes (odl.space.base_ntuples.NtuplesBaseVector attribute), 218
nbytes (odl.space.cu_ntuples.CudaFnVector attribute), 236
nbytes (odl.space.cu_ntuples.CudaNtuplesVector attribute), 245
nbytes (odl.space.ntuples.FnVector attribute), 281
nbytes (odl.space.ntuples.NtuplesVector attribute), 292
ndim (odl.discr.discretization.DiscretizationVector attribute), 67
ndim (odl.discr.discretization.RawDiscretizationVector attribute), 71
ndim (odl.discr.grid.RegularGrid attribute), 75
ndim (odl.discr.grid.TensorGrid attribute), 84
ndim (odl.discr.lp_discr.DiscreteLp attribute), 94
ndim (odl.discr.lp_discr.DiscreteLpVector attribute), 97
ndim (odl.set.domain.IntervalProd attribute), 158
ndim (odl.space.base_ntuples.FnBaseVector attribute), 212
ndim (odl.space.base_ntuples.NtuplesBaseVector attribute), 218
ndim (odl.space.cu_ntuples.CudaFnVector attribute), 236
ndim (odl.space.cu_ntuples.CudaNtuplesVector attribute), 245
ndim (odl.space.ntuples.FnVector attribute), 281
ndim (odl.space.ntuples.NtuplesVector attribute), 292
NearestInterpolation (class in odl.discr.discr_mappings), 45
negative() (odl.util.ufuncs.CudaNtuplesUFuncs method), 329
negative() (odl.util.ufuncs.DiscreteLpUFuncs method), 342
negative() (odl.util.ufuncs.NtuplesBaseUFuncs method), 355
negative() (odl.util.ufuncs.NtuplesUFuncs method), 367
negative() (odl.util.ufuncs.ProductSpaceUFuncs method), 380
newtons_method() (in module odl.solvers.vector.newton), 207
nextafter() (odl.util.ufuncs.CudaNtuplesUFuncs method), 329
nextafter() (odl.util.ufuncs.DiscreteLpUFuncs method), 342
nextafter() (odl.util.ufuncs.NtuplesBaseUFuncs method), 355
nextafter() (odl.util.ufuncs.NtuplesUFuncs method), 368
nextafter() (odl.util.ufuncs.ProductSpaceUFuncs method), 380
norm (odl.space.ntuples.FnCustomNorm attribute), 273
norm() (odl.diagnostics.operator.OperatorTest method), 34
norm() (odl.diagnostics.space.SpaceTest method), 37
norm() (odl.set.space.LinearSpace method), 186
norm() (odl.set.space.LinearSpaceVector method), 189
norm() (odl.set.space.UniversalSpace method), 193
norm() (odl.space.base_ntuples.FnWeightingBase method), 215
norm() (odl.space.cu_ntuples.CudaFnConstWeighting method), 229
norm() (odl.space.cu_ntuples.CudaFnCustomDist method), 230
norm() (odl.space.cu_ntuples.CudaFnCustomNorm method), 233
norm() (odl.space.cu_ntuples.CudaFnNoWeighting method), 235
norm() (odl.space.cu_ntuples.CudaFnVectorWeighting method), 242
norm() (odl.space.ntuples.FnConstWeighting method), 270
norm() (odl.space.ntuples.FnCustomDist method), 272
norm() (odl.space.ntuples.FnMatrixWeighting method), 276
norm() (odl.space.ntuples.FnNoWeighting method), 278
norm() (odl.space.ntuples.FnVectorWeighting method), 287
not_equal() (odl.util.ufuncs.CudaNtuplesUFuncs method), 329
not_equal() (odl.util.ufuncs.DiscreteLpUFuncs method), 342
not_equal() (odl.util.ufuncs.NtuplesBaseUFuncs method), 355
not_equal() (odl.util.ufuncs.NtuplesUFuncs method), 368
not_equal() (odl.util.ufuncs.ProductSpaceUFuncs method), 380
ntotal (odl.discr.grid.RegularGrid attribute), 75
ntotal (odl.discr.grid.TensorGrid attribute), 84
ntuple (odl.discr.discretization.DiscretizationVector attribute), 67
ntuple (odl.discr.discretization.RawDiscretizationVector attribute), 71
ntuple (odl.discr.lp_discr.DiscreteLpVector attribute), 97
Ntuples (class in odl.space.ntuples), 289
NtuplesBase (class in odl.space.base_ntuples), 216
NtuplesBaseUFuncs (class in odl.util.ufuncs), 345
NtuplesBaseVector (class in odl.space.base_ntuples), 218
NtuplesUFuncs (class in odl.util.ufuncs), 358
NtuplesVector (class in odl.space.ntuples), 291

O

one() (odl.discr.discretization.Discretization method), 65

one() (odl.set.pspace.ProductSpace method), 168
one() (odl.set.space.LinearSpace method), 186
one() (odl.set.space.UniversalSpace method), 194
one() (odl.space.base_ntuples.FnBase method), 211
one() (odl.space.cu_ntuples.CudaFn method), 226
one() (odl.space.ntuples.Fn method), 267
OpDomainError, 124
Operator (class in odl.operator.operator), 124
OperatorComp (class in odl.operator.operator), 127
OperatorLeftScalarMult (class in odl.operator.operator), 130
OperatorLeftVectorMult (class in odl.operator.operator), 132
OperatorPointwiseProduct (class in odl.operator.operator), 135
OperatorRightScalarMult (class in odl.operator.operator), 137
OperatorRightVectorMult (class in odl.operator.operator), 140
OperatorSum (class in odl.operator.operator), 142
OperatorTest (class in odl.diagnostics.operator), 33
OpNotImplementedError, 124
OpRangeError, 124
OpTypeError, 124
order (odl.discr.discr_mappings.FunctionSetMapping attribute), 39
order (odl.discr.discr_mappings.GridCollocation attribute), 41
order (odl.discr.discr_mappings.LinearInterpolation attribute), 44
order (odl.discr.discr_mappings.NearestInterpolation attribute), 46
order (odl.discr.discr_mappings.RawGridCollocation attribute), 49
order (odl.discr.discr_mappings.RawNearestInterpolation attribute), 52
order (odl.discr.grid.RegularGrid attribute), 75
order (odl.discr.grid.TensorGrid attribute), 84
order (odl.discr.lp_discr.DiscreteLp attribute), 94

P

Partial (class in odl.solvers.util.partial), 205
parts (odl.set.pspace.ProductSpaceVector attribute), 170
points() (odl.discr.grid.RegularGrid method), 81
points() (odl.discr.grid.TensorGrid method), 89
points() (odl.discr.lp_discr.DiscreteLp method), 95
power() (odl.util.ufuncs.CudaNtuplesUFuncs method), 329
power() (odl.util.ufuncs.DiscreteLpUFuncs method), 342
power() (odl.util.ufuncs.NtuplesBaseUFuncs method), 355
power() (odl.util.ufuncs.NtuplesUFuncs method), 368
power() (odl.util.ufuncs.ProductSpaceUFuncs method), 381
power_method_opnorm() (in module odl.operator.oputils), 147
PrintIterationPartial (class in odl.solvers.util.partial), 206
PrintStatusPartial (class in odl.solvers.util.partial), 206
prod() (odl.util.ufuncs.CudaNtuplesUFuncs method), 329
prod() (odl.util.ufuncs.DiscreteLpUFuncs method), 342
prod() (odl.util.ufuncs.NtuplesBaseUFuncs method), 355
prod() (odl.util.ufuncs.NtuplesUFuncs method), 368
prod() (odl.util.ufuncs.ProductSpaceUFuncs method), 381
ProductSpace (class in odl.set.pspace), 165
ProductSpaceOperator (class in odl.operator.pspace_ops), 153
ProductSpaceUFuncs (class in odl.util.ufuncs), 371
ProductSpaceVector (class in odl.set.pspace), 169
ProgressBar (class in odl.util.testutils), 316
ProgressRange (class in odl.util.testutils), 317
pywt_coeff_to_array() (in module odl.trafos.wavelet), 311

R

rad2deg() (odl.util.ufuncs.CudaNtuplesUFuncs method), 330
rad2deg() (odl.util.ufuncs.DiscreteLpUFuncs method), 342
rad2deg() (odl.util.ufuncs.NtuplesBaseUFuncs method), 355
rad2deg() (odl.util.ufuncs.NtuplesUFuncs method), 368
rad2deg() (odl.util.ufuncs.ProductSpaceUFuncs method), 381
range (odl.discr.discr_mappings.FunctionSetMapping attribute), 39
range (odl.discr.discr_mappings.GridCollocation attribute), 41
range (odl.discr.discr_mappings.LinearInterpolation attribute), 44
range (odl.discr.discr_mappings.NearestInterpolation attribute), 46
range (odl.discr.discr_mappings.RawGridCollocation attribute), 49
range (odl.discr.discr_mappings.RawNearestInterpolation attribute), 52
range (odl.discr.discr_ops.DiscreteDivergence attribute), 54
range (odl.discr.discr_ops.DiscreteGradient attribute), 56
range (odl.discr.discr_ops.DiscretePartDeriv attribute), 59
range (odl.operator.default_ops.ConstantOperator attribute), 103
range (odl.operator.default_ops.IdentityOperator attribute), 105
range (odl.operator.default_ops.InnerProductAdjointOperator attribute), 107
range (odl.operator.default_ops.InnerProductOperator attribute), 110

range (odl.operator.default_ops.LinCombOperator attribute), 111
range (odl.operator.default_ops.MultiplyOperator attribute), 113
range (odl.operator.default_ops.ResidualOperator attribute), 115
range (odl.operator.default_ops.ScalingOperator attribute), 117
range (odl.operator.default_ops.ZeroOperator attribute), 119
range (odl.operator.operator.FunctionalLeftVectorMult attribute), 122
range (odl.operator.operator.Operator attribute), 126
range (odl.operator.operator.OperatorComp attribute), 128
range (odl.operator.operator.OperatorLeftScalarMult attribute), 131
range (odl.operator.operator.OperatorLeftVectorMult attribute), 133
range (odl.operator.operator.OperatorPointwiseProduct attribute), 136
range (odl.operator.operator.OperatorRightScalarMult attribute), 138
range (odl.operator.operator.OperatorRightVectorMult attribute), 141
range (odl.operator.operator.OperatorSum attribute), 143
range (odl.operator.pspace_ops.ComponentProjection attribute), 149
range (odl.operator.pspace_ops.ComponentProjectionAdjoint attribute), 151
range (odl.operator.pspace_ops.ProductSpaceOperator attribute), 155
range (odl.space.fspace.FunctionSet attribute), 252
range (odl.space.fspace.FunctionSetVector attribute), 254
range (odl.space.fspace.FunctionSpace attribute), 255
range (odl.space.fspace.FunctionSpaceVector attribute), 259
range (odl.space.ntuples.MatVecOperator attribute), 289
range (odl.tomo.backends.stir_bindings.BackProjectorByBinWrapper attribute), 301
range (odl.tomo.backends.stir_bindings.ForwardProjectorByBinWrapper attribute), 303
range (odl.trafos.wavelet.DiscreteWaveletTransform attribute), 307
range (odl.trafos.wavelet.DiscreteWaveletTransformInverse attribute), 308
RawDiscretization (class in odl.discr.discretization), 68
RawDiscretizationVector (class in odl.discr.discretization), 70
RawGridCollocation (class in odl.discr.discr_mappings), 48
RawNearestInterpolation (class in odl.discr.discr_mappings), 51
real (odl.space.ntuples.FnVector attribute), 281
real_dtype (odl.discr.discretization.Discretization attribute), 63
real_dtype (odl.discr.lp_discr.DiscreteLp attribute), 94
real_dtype (odl.space.base_ntuples.FnBase attribute), 209
real_dtype (odl.space.cu_ntuples.CudaFn attribute), 221
real_dtype (odl.space.ntuples.Fn attribute), 262
RealNumbers (class in odl.set.sets), 177
reciprocal() (odl.util.ufuncs.CudaNtuplesUFuncs method), 330
reciprocal() (odl.util.ufuncs.DiscreteLpUFuncs method), 343
reciprocal() (odl.util.ufuncs.NtuplesBaseUFuncs method), 355
reciprocal() (odl.util.ufuncs.NtuplesUFuncs method), 368
reciprocal() (odl.util.ufuncs.ProductSpaceUFuncs method), 381
Rectangle() (in module odl.set.domain), 165
RegularGrid (class in odl.discr.grid), 73
remainder() (odl.util.ufuncs.CudaNtuplesUFuncs method), 330
remainder() (odl.util.ufuncs.DiscreteLpUFuncs method), 343
remainder() (odl.util.ufuncs.NtuplesBaseUFuncs method), 356
remainder() (odl.util.ufuncs.NtuplesUFuncs method), 368
remainder() (odl.util.ufuncs.ProductSpaceUFuncs method), 381
ResidualOperator (class in odl.operator.default_ops), 114
restriction (odl.discr.discretization.Discretization attribute), 63
restriction (odl.discr.discretization.RawDiscretization attribute), 69
restriction (odl.discr.lp_discr.DiscreteLp attribute), 94
right_shift() (odl.util.ufuncs.CudaNtuplesUFuncs method), 330
right_shift() (odl.util.ufuncs.DiscreteLpUFuncs method), 343
right_shift() (odl.util.ufuncs.NtuplesBaseUFuncs method), 356
right_shift() (odl.util.ufuncs.NtuplesUFuncs method), 369
right_shift() (odl.util.ufuncs.ProductSpaceUFuncs method), 381
rint() (odl.util.ufuncs.CudaNtuplesUFuncs method), 330
rint() (odl.util.ufuncs.DiscreteLpUFuncs method), 343
rint() (odl.util.ufuncs.NtuplesBaseUFuncs method), 356
rint() (odl.util.ufuncs.NtuplesUFuncs method), 369
rint() (odl.util.ufuncs.ProductSpaceUFuncs method), 381
Rn() (in module odl.space.ntuples), 297
run_tests() (odl.diagnostics.operator.OperatorTest method), 34
run_tests() (odl.diagnostics.space.SpaceTest method), 37

S

samples() (in module odl.diagnostics.examples), 32
scalar_examples() (in module odl.diagnostics.examples), 32
ScalingOperator (class in odl.operator.default_ops), 116
self_adjoint() (odl.diagnostics.operator.OperatorTest method), 34
send() (odl.solvers.util.partial.ForEachPartial method), 205
send() (odl.solvers.util.partial.Partial method), 205
send() (odl.solvers.util.partial.PrintIterationPartial method), 206
send() (odl.solvers.util.partial.PrintStatusPartial method), 206
send() (odl.solvers.util.partial.StorePartial method), 206
Set (class in odl.set.sets), 178
set_zero() (odl.set.space.LinearSpaceVector method), 189
sets (odl.set.sets.CartesianProduct attribute), 173
shape (odl.discr.discretization.DiscretizationVector attribute), 67
shape (odl.discr.discretization.RawDiscretizationVector attribute), 71
shape (odl.discr.grid.RegularGrid attribute), 75
shape (odl.discr.grid.TensorGrid attribute), 84
shape (odl.discr.lp_discr.DiscreteLp attribute), 95
shape (odl.discr.lp_discr.DiscreteLpVector attribute), 97
shape (odl.space.base_ntuples.FnBaseVector attribute), 212
shape (odl.space.base_ntuples.NtuplesBaseVector attribute), 218
shape (odl.space.cu_ntuples.CudaFnVector attribute), 236
shape (odl.space.cu_ntuples.CudaNtuplesVector attribute), 245
shape (odl.space.ntuples.FnVector attribute), 281
shape (odl.space.ntuples.NtuplesVector attribute), 292
shepp_logan() (in module odl.util.phantom), 315
show() (odl.discr.lp_discr.DiscreteLpVector method), 99
show() (odl.set.pspace.ProductSpaceVector method), 172
show_discrete_function() (in module odl.util.graphics), 314
sign() (odl.util.ufuncs.CudaNtuplesUFuncs method), 330
sign() (odl.util.ufuncs.DiscreteLpUFuncs method), 343
sign() (odl.util.ufuncs.NtuplesBaseUFuncs method), 356
sign() (odl.util.ufuncs.NtuplesUFuncs method), 369
sign() (odl.util.ufuncs.ProductSpaceUFuncs method), 382
signbit() (odl.util.ufuncs.CudaNtuplesUFuncs method), 330
signbit() (odl.util.ufuncs.DiscreteLpUFuncs method), 343
signbit() (odl.util.ufuncs.NtuplesBaseUFuncs method), 356
signbit() (odl.util.ufuncs.NtuplesUFuncs method), 369
signbit() (odl.util.ufuncs.ProductSpaceUFuncs method), 382
simple_operator() (in module odl.operator.operator), 146
sin() (odl.util.ufuncs.CudaNtuplesUFuncs method), 331
sin() (odl.util.ufuncs.DiscreteLpUFuncs method), 343
sin() (odl.util.ufuncs.NtuplesBaseUFuncs method), 356
sin() (odl.util.ufuncs.NtuplesUFuncs method), 369
sin() (odl.util.ufuncs.ProductSpaceUFuncs method), 382
sinh() (odl.util.ufuncs.CudaNtuplesUFuncs method), 331
sinh() (odl.util.ufuncs.DiscreteLpUFuncs method), 344
sinh() (odl.util.ufuncs.NtuplesBaseUFuncs method), 356
sinh() (odl.util.ufuncs.NtuplesUFuncs method), 369
sinh() (odl.util.ufuncs.ProductSpaceUFuncs method), 382
size (odl.discr.discretization.Discretization attribute), 64
size (odl.discr.discretization.DiscretizationVector attribute), 67
size (odl.discr.discretization.RawDiscretization attribute), 69
size (odl.discr.discretization.RawDiscretizationVector attribute), 71
size (odl.discr.lp_discr.DiscreteLp attribute), 95
size (odl.discr.lp_discr.DiscreteLpVector attribute), 97
size (odl.set.domain.IntervalProd attribute), 158
size (odl.set.pspace.ProductSpace attribute), 166
size (odl.set.pspace.ProductSpaceVector attribute), 170
size (odl.space.base_ntuples.FnBase attribute), 209
size (odl.space.base_ntuples.FnBaseVector attribute), 212
size (odl.space.base_ntuples.NtuplesBase attribute), 216
size (odl.space.base_ntuples.NtuplesBaseVector attribute), 218
size (odl.space.cu_ntuples.CudaFn attribute), 222
size (odl.space.cu_ntuples.CudaFnVector attribute), 236
size (odl.space.cu_ntuples.CudaNtuples attribute), 242
size (odl.space.cu_ntuples.CudaNtuplesVector attribute), 245
size (odl.space.ntuples.Fn attribute), 262
size (odl.space.ntuples.FnVector attribute), 281
size (odl.space.ntuples.Ntuples attribute), 290
size (odl.space.ntuples.NtuplesVector attribute), 292
size() (odl.discr.grid.RegularGrid method), 82
size() (odl.discr.grid.TensorGrid method), 90
skip_if_no_cuda() (in module odl.util.testutils), 318
skip_if_no_pywavelets() (in module odl.util.testutils), 318
space (odl.discr.discretization.DiscretizationVector attribute), 67
space (odl.discr.discretization.RawDiscretizationVector attribute), 71
space (odl.discr.lp_discr.DiscreteLpVector attribute), 97
space (odl.set.pspace.ProductSpaceVector attribute), 170
space (odl.set.space.LinearSpaceVector attribute), 188
space (odl.space.base_ntuples.FnBaseVector attribute), 212
space (odl.space.base_ntuples.NtuplesBaseVector attribute), 219
space (odl.space.cu_ntuples.CudaFnVector attribute), 237

space (odl.space.cu_ntuples.CudaNtuplesVector attribute), 245
space (odl.space.fspace.FunctionSetVector attribute), 254
space (odl.space.fspace.FunctionSpaceVector attribute), 259
space (odl.space.ntuples.FnVector attribute), 281
space (odl.space.ntuples.NtuplesVector attribute), 292
spaces (odl.set.pspace.ProductSpace attribute), 166
SpaceTest (class in odl.diagnostics.space), 35
sparse_meshgrid() (in module odl.discr.grid), 91
sqrt() (odl.util.ufuncs.CudaNtuplesUFuncs method), 331
sqrt() (odl.util.ufuncs.DiscreteLpUFuncs method), 344
sqrt() (odl.util.ufuncs.NtuplesBaseUFuncs method), 357
sqrt() (odl.util.ufuncs.NtuplesUFuncs method), 369
sqrt() (odl.util.ufuncs.ProductSpaceUFuncs method), 382
square() (odl.util.ufuncs.CudaNtuplesUFuncs method), 331
square() (odl.util.ufuncs.DiscreteLpUFuncs method), 344
square() (odl.util.ufuncs.NtuplesBaseUFuncs method), 357
square() (odl.util.ufuncs.NtuplesUFuncs method), 370
square() (odl.util.ufuncs.ProductSpaceUFuncs method), 382
squeeze() (odl.discr.grid.RegularGrid method), 82
squeeze() (odl.discr.grid.TensorGrid method), 90
squeeze() (odl.set.domain.IntervalProd method), 163
start() (odl.util.testutils.ProgressBar method), 317
steepest_descent() (in module odl.solvers.scalar.gradient), 200
StepLength (class in odl.solvers.scalar.steplen), 204
stir_projector_from_file() (in module odl.tomo.backends.stir_bindings), 304
StirVerbosity (class in odl.tomo.backends.stir_bindings), 304
StorePartial (class in odl.solvers.util.partial), 206
stride (odl.discr.grid.RegularGrid attribute), 75
Strings (class in odl.set.sets), 180
subtract() (odl.util.ufuncs.CudaNtuplesUFuncs method), 331
subtract() (odl.util.ufuncs.DiscreteLpUFuncs method), 344
subtract() (odl.util.ufuncs.NtuplesBaseUFuncs method), 357
subtract() (odl.util.ufuncs.NtuplesUFuncs method), 370
subtract() (odl.util.ufuncs.ProductSpaceUFuncs method), 382
sum() (odl.util.ufuncs.CudaNtuplesUFuncs method), 331
sum() (odl.util.ufuncs.DiscreteLpUFuncs method), 344
sum() (odl.util.ufuncs.NtuplesBaseUFuncs method), 357
sum() (odl.util.ufuncs.NtuplesUFuncs method), 370
sum() (odl.util.ufuncs.ProductSpaceUFuncs method), 383

T

T (odl.discr.discretization.DiscretizationVector attribute), 66
T (odl.discr.lp_discr.DiscreteLpVector attribute), 96
T (odl.operator.default_ops.InnerProductOperator attribute), 109
T (odl.set.pspace.ProductSpaceVector attribute), 170
T (odl.set.space.LinearSpaceVector attribute), 187
T (odl.space.base_ntuples.FnBaseVector attribute), 211
T (odl.space.cu_ntuples.CudaFnVector attribute), 235
T (odl.space.cu_ntuples.CudaNtuplesVector attribute), 244
T (odl.space.fspace.FunctionSpaceVector attribute), 258
T (odl.space.ntuples.FnVector attribute), 279
tan() (odl.util.ufuncs.CudaNtuplesUFuncs method), 331
tan() (odl.util.ufuncs.DiscreteLpUFuncs method), 344
tan() (odl.util.ufuncs.NtuplesBaseUFuncs method), 357
tan() (odl.util.ufuncs.NtuplesUFuncs method), 370
tan() (odl.util.ufuncs.ProductSpaceUFuncs method), 383
tanh() (odl.util.ufuncs.CudaNtuplesUFuncs method), 332
tanh() (odl.util.ufuncs.DiscreteLpUFuncs method), 344
tanh() (odl.util.ufuncs.NtuplesBaseUFuncs method), 357
tanh() (odl.util.ufuncs.NtuplesUFuncs method), 370
tanh() (odl.util.ufuncs.ProductSpaceUFuncs method), 383
TensorGrid (class in odl.discr.grid), 83
timeit() (in module odl.util.testutils), 319
Timer (class in odl.util.testutils), 317
true_divide() (odl.util.ufuncs.CudaNtuplesUFuncs method), 332
true_divide() (odl.util.ufuncs.DiscreteLpUFuncs method), 345
true_divide() (odl.util.ufuncs.NtuplesBaseUFuncs method), 357
true_divide() (odl.util.ufuncs.NtuplesUFuncs method), 370
true_divide() (odl.util.ufuncs.ProductSpaceUFuncs method), 383
true_ndim (odl.set.domain.IntervalProd attribute), 158
trunc() (odl.util.ufuncs.CudaNtuplesUFuncs method), 332
trunc() (odl.util.ufuncs.DiscreteLpUFuncs method), 345
trunc() (odl.util.ufuncs.NtuplesBaseUFuncs method), 358
trunc() (odl.util.ufuncs.NtuplesUFuncs method), 370
trunc() (odl.util.ufuncs.ProductSpaceUFuncs method), 383

U

ufunc (odl.discr.discretization.DiscretizationVector attribute), 67
ufunc (odl.discr.discretization.RawDiscretizationVector attribute), 72
ufunc (odl.discr.lp_discr.DiscreteLpVector attribute), 97
ufunc (odl.set.pspace.ProductSpaceVector attribute), 170

ufunc (odl.space.base_ntuples.FnBaseVector attribute), 212
ufunc (odl.space.base_ntuples.NtuplesBaseVector attribute), 219
ufunc (odl.space.cu_ntuples.CudaFnVector attribute), 237
ufunc (odl.space.cu_ntuples.CudaNtuplesVector attribute), 245
ufunc (odl.space.ntuples.FnVector attribute), 282
ufunc (odl.space.ntuples.NtuplesVector attribute), 292
uniform_discr() (in module odl.discr.lp_discr), 100
uniform_discr_fromspace() (in module odl.discr.lp_discr), 101
uniform_sampling() (in module odl.discr.grid), 91
UniversalSet (class in odl.set.sets), 181
UniversalSpace (class in odl.set.space), 190
update() (odl.util.testutils.ProgressBar method), 317
uspace (odl.discr.discretization.Discretization attribute), 64
uspace (odl.discr.discretization.RawDiscretization attribute), 69
uspace (odl.discr.lp_discr.DiscreteLp attribute), 95

V

vector (odl.space.cu_ntuples.CudaFnVectorWeighting attribute), 240
vector (odl.space.ntuples.FnVectorWeighting attribute), 286
vector() (in module odl.space.space_utils), 299
vector() (odl.diagnostics.space.SpaceTest method), 37
vector_assign() (odl.diagnostics.space.SpaceTest method), 37
vector_copy() (odl.diagnostics.space.SpaceTest method), 37
vector_equals() (odl.diagnostics.space.SpaceTest method), 38
vector_examples() (in module odl.diagnostics.examples), 33
vector_is_valid() (odl.space.cu_ntuples.CudaFnVectorWeighting method), 242
vector_is_valid() (odl.space.ntuples.FnVectorWeighting method), 287
vector_set_zero() (odl.diagnostics.space.SpaceTest method), 38
vector_space() (odl.diagnostics.space.SpaceTest method), 38
volume (odl.set.domain.IntervalProd attribute), 158

W

wavelet_decomposition3d() (in module odl.trafos.wavelet), 312
wavelet_reconstruction3d() (in module odl.trafos.wavelet), 313
weighted_dist() (in module odl.space.ntuples), 297
weighted_inner() (in module odl.space.ntuples), 298
weighted_norm() (in module odl.space.ntuples), 298
weights (odl.set.pspace.ProductSpace attribute), 166
with_metaclass() (in module odl.util.utility), 386
wrap_reduction_base() (in module odl.util.ufuncs), 384
wrap_reduction_discretelp() (in module odl.util.ufuncs), 384
wrap_reduction_productspace() (in module odl.util.ufuncs), 384
wrap_ufunc_base() (in module odl.util.ufuncs), 384
wrap_ufunc_discretelp() (in module odl.util.ufuncs), 384
wrap_ufunc_ntuples() (in module odl.util.ufuncs), 384
wrap_ufunc_productspace() (in module odl.util.ufuncs), 384

Z

zero() (odl.discr.discretization.Discretization method), 65
zero() (odl.set.pspace.ProductSpace method), 168
zero() (odl.set.space.LinearSpace method), 186
zero() (odl.set.space.UniversalSpace method), 194
zero() (odl.space.base_ntuples.FnBase method), 211
zero() (odl.space.cu_ntuples.CudaFn method), 226
zero() (odl.space.fspace.FunctionSpace method), 257
zero() (odl.space.ntuples.Fn method), 267
ZeroOperator (class in odl.operator.default_ops), 118